Parallelizing the Cluster Rank Analysis application

Anthony G. Esposito Jr.

Parallelizing the Cluster Rank Analysis Application

by

Anthony G. Esposito, Jr.

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

in

Bioinformatics

at the

Rochester Institute of Technology

Thesis Advisor:

Dr. Michael Osier

Thesis Committee Members:

Dr. Dina Newman, Dr. Paul Shipman

bioinformatics at RIT

Rochester Institute of Technology
Department of Biological Sciences
Bioinformatics Program

To: Head, Department of Biological Sciences

The undersigned state that Anthony G. Esposito, Jr., a candidate for the Master of Science degree in Bioinformatics, has submitted his/her thesis and has satisfactorily defended it.

This completes the requirements for the Master of Science degree in Bioinformatics at Rochester Institute of Technology.

Thesis committee members:

Name	**Date**
Michael Osier Dina Newman (Committee Chair)	5/24/07
Dina Newman (Thesis Advisor)	5/24/07
Paul Shipman	5/24/07

Gary R. Skuse, Ph.D.
Director of Bioinformatics

Abstract

A wide range of researchers is beginning to utilize customized statistical methods for analyzing data as hardware and software become cheaper and more widely available. Cluster Rank Analysis (CRA) is an existing multivariate statistical algorithm that existed as an inefficient service-oriented application. Here it is described how CRA was optimized and parallelized using an available computing cluster and both open source and custom software. This was followed by the development of a command-line submission system for CRA jobs, as well as a Web retrieval system for the results of analyses. A subsequent timing study revealed speedup that quickly rose to 15 by the use 35 processors, and should reach a proposed maximum of 19 given over 100 processors. It was found that this speedup was limited primarily by the serial portion of code; the Ethernet communication network was sufficient for this application. By the time that even 10 processors were involved in parallel runs, the average runtime had dropped from over 100 minutes to approximately 15 minutes, before being reduced to 6 minutes by 80 processors. The locations of bottlenecks suggest that further performance increases are possible through additional parallelization. This work with CRA illustrates (1) the speed with which high-performance in-house applications can be developed and (2) the speed and efficiency with which statistical analyses of complex data structures can be carried out given commodity hardware and software resources.

Table of Contents

List of Figures v

List of Tables vi

Acknowledgements vii

Introduction 1

Materials & Methods 18

Results 31

Discussion 47

Conclusions 61

References 64

List of Figures

Figure 1	Diagram of a mitochondrion, in the context of the eukaryotic cell in which it and others like it are found.	4
Figure 2	A simplified example of the CRA algorithm.	8
Figure 3	Approximate computation time to permute a set of elements containing an arbitrary number of data points.	12
Figure 4	Schematics showing two possible physical architectures for CRA instances.	14
Figure 5	Graph showing the effect of Amdahl's Law on maximum attainable speedup, which is limited by the fraction of a program that must run sequentially.	27
Figure 6	Graph showing the reduction in runtime as the number of permutation-performing processors is increased.	35
Figure 7	Graph showing the increase in speedup as the number of permutation-performing processors is increased.	38
Figure 8	Graph showing the decrease in efficiency as the number of permutation-performing processors is increased.	41
Figure 9	Graph showing the region of super-linear speedup for the two timing runs.	42
Figure 10	Graph overlaying Figure 6 and Figure 7, allowing the correlation between runtime and speedup to be clearly seen.	43
Figure 11	Graph overlaying Figure 7 and Figure 8, allowing the correlation between speedup and efficiency to be seen.	44
Figure 12	Screenshot of a typical results page for a CRA run.	46

List of Tables

Table 1	The number of processors used and total runtimes for the first and second series of CRA runs in parallel.	34
Table 2	The number of processors used and permutation-based runtimes for the first and second series of CRA runs in parallel.	36
Table 3	The number of processors used, overall speedup values, and permutation-based speedup values for the first and second series of CRA runs in parallel.	37
Table 4	The number of processors used, overall efficiency values, and permutation-based efficiency values for the first and second series of CRA runs in parallel.	40
Table 5	The number of processors used, and overall Karp-Flatt values for the first and second series of timing runs in parallel.	45

Acknowledgements

Introduction

Many problems in biology, among other fields, exist alongside relevant sets of data. Very often these sets contain large amounts of data, possibly with multiple variables and some sort of inherent structure. One type of analysis that could be performed in such cases would allow data from one dimension to give some sort of meaning to data that exists in one, two, or even more dimensions. A tree representing some sort of hierarchical data gives two forms of meaning, represented by the two dimensions the tree occupies. This tree is capable of yielding even more information about the data of which it is constructed. It can be rearranged and have its leaves and nodes in a different order at each level of the tree; however, as long as all of the edges between two points are maintained, with no new edges or nodes being created and no edges or nodes being destroyed, the tree remains the same.

Since a tree can thus be rearranged without affecting the integrity of the tree, secondary sets of variables can be used to guide these rearrangements and give them meaning. Applying a secondary data set to the tree could be done by sorting the leaves while maintaining the topology of the tree; for example, the leaves could be rearranged and sorted such that a gradient of secondary values is developed across the leaves. In order to determine the proper order of sets of leaves that join at higher internal nodes, each internal node could be represented by the mean value of all the child nodes. The terminal child nodes would of course be the leaves themselves, whose data points would be the basis for all internal node values.

In order to give meaning to the rearranged leaves, and determine how strong a gradient is created, scores like the Spearman rank order coefficient could be calculated. In this case, the numeric order of the leaves is compared to the fully sorted numeric order of the leaves; the latter assumes the topology of the tree has been broken. The gradient would then be easily quantified by a numeric value.

The potential to apply such an analysis to structured, multivariate data has potential in many different fields of research. Two areas that are already using this type of analysis, known as Cluster Rank Analysis (CRA), are human genetics and ecology. The application of this type of analysis is especially relevant in biological fields, as the use of hierarchical structures such as phylogenetic trees is widespread.

Analyzing Human Genetics Data: Presbycusis

Age-related hearing loss, also known as presbycusis, is one of the most common age-related ailments that affect humans. It is estimated that 30-35% of the population between the ages of 65 and 75 suffers from some degree of hearing loss; above the age of 75, the estimated percentage of the population that experiences a decrease in hearing ability jumps to 40-50% (1, 2). The hearing loss experienced with presbycusis begins in higher frequencies towards the top of the audible range for humans. As the condition worsens, however, the loss of frequency differentiation begins to occur in progressively deepening frequencies (3, 4). This can become particularly debilitating as the lower frequencies that humans communicate at begin to be included in the range of loss, threatening the ability to distinguish speech as the disorder worsens (4).

Pedigree analysis has uncovered a trend in presbycusis sufferers: presbycusis tends to be more prevalent in the children of a presbycusic mother than in children of a presbycusic father, assuming that the other parent in each case is non-presbycusic (5). It is thus hypothesized that presbycusis is linked to the mitochondrial genome and not the nuclear genome, due to the egg, but not the sperm, contributing the mitochondria to an embryo (6). There are additional disorders that can be found in OMIM that are linked to the mitochondria, either hypothetically or by actual mitochondrial gene linkage (5, 7).

Mitochondria, organelles present in most eukaryotic cells, are the locations at which oxidative phosphorylation occurs during cellular respiration. The mitochondrion has two lipid bilayer membranes; the outer membrane encases the organelle, while the inner membrane surrounds a convoluted interior matrix. Oxidative phosphorylation occurs across this inner membrane, with ATP synthase converting ADP to ATP as the end result. The mitochondrion is thought to have evolved from an intracellular endosymbiotic relationship earlier in eukaryotic evolution; it has its own circular genome, complete with an origin of replication, and reproduces by binary fission. Both of these traits are hallmark characteristics of prokaryotic cells (8). The circular genome encodes, among other things, the enzymes needed to replicate and sustain it (8). The structure of a single mitochondrion, including the inner and outer membranes, cristae (internal folds), and matrix, is presented in Figure 1.

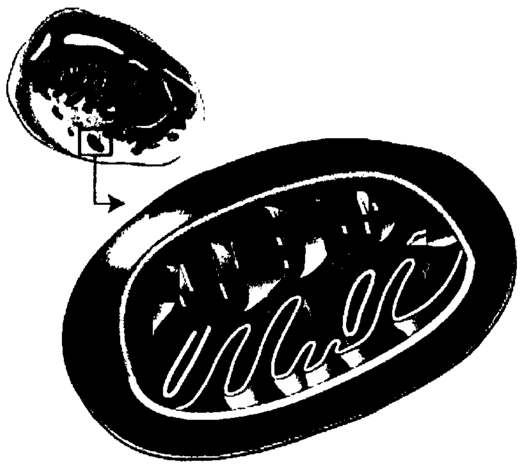


Figure 1. Diagram of a mitochondrion, in the context of the eukaryotic cell in which it and others like it are found. Note that the circular mitochondrial genome is found in the innermost space of the organelle, known as the matrix.

The origin of replication on the mitochondrial genome is flanked by two hypervariable regions (HVR's); these regions are comprised of repeats in the DNA, and are highly polymorphic (9). These HVR's are named HVR-I and HVR-II; HVR-I precedes the origin of replication, while HVR-II immediately follows the origin of replication (10). Currently, the correlation between specific single nucleotide polymorphisms (SNPs) occurring in HVR-II and onset and degree of presbycusis is a topic of much interest to human geneticists; for example, researchers are sequencing the mitochondrial HVR-II of volunteers, as well as performing a series of standard auditory tests (11). These data can then used for correlative statistical analyses.

Analyzing Ecological Data: Environmental Sampling

Over the past century, the impact of humans on the environment has been tremendous. Effects ranging from those of global warming, to waste disposal, to forest clear-cutting indirectly and directly influence the distribution of species across the landscape. There may be a trend of endangerment and extinction due to human factors; one group of organisms that can be used to measure such impacts is the order Anura (12). This order, which includes frogs and toads, is a diverse group found not only in every continent except Antarctica, but also in ecosystems ranging from mountain forests to runoff-water collection ponds (13). Such a widespread and robust distribution makes this group particularly useful in the determination of the effects of humans on animal populations (14).

The branch of research that studies the management of forests and the balance between environmental and societal forest needs is known as silviculture; while it is easy to measure the societal needs of forest use by examining human needs such as demand for lumber, it is much more difficult to measure the environmental needs (15). Thus, researchers can use organisms such as anurans to gauge the impact of not only silvicultural issues, but also of other human development issues such as transportation requirements, for example.

While the human impact on the environment may influence the distribution of anuran species, there are other factors to consider in examining the distribution of these amphibians over a given period of time (14). One group of factors consists of those occurring naturally, such as the type of vegetation in a given plot, the percentage of water, rock, or leaf litter cover, or the elevation (16).

Another group of factors consists of the other anuran species at a plot. Thus, not only are the relationships between a species and the local environmental factors examined, but inter-species relationships are also considered (14).

The factors that are considered are typically measured in the field. In one study, plots 20m in diameter were made in several replicate sites in the Midwest United States. Measurements and observations were made of various variables at the sites over the course of several months (14). Of note, not all variables were continuous or discrete values collected via measurements such as percentages. There were categorical variables as well, such as local forest type.

It is important to realize that while the common perception is that development by humans causes the distribution of species to change and consequently be reduced, that is not always the case. By collecting data from the field and analyzing it using statistical methods, new correlations can be discovered; for example, an increase in the number of a given species in clear-cut forests or near roads might not be expected, but would become evident in a detailed analysis of the data. One potential relationship that might be discoverable through further analysis is that cohabitation relationships with regard to a specific set of environmental factors might vary depending upon those factors.

Common Traits to both Biological Cases

The above cases share a number of characteristics. Both have large data sets; for the presbycusis sets, there are DNA sequence data and auditory test results data. The environmental sampling sets consist of data on species count per plot, and environmental factor data. Additionally, both cases feature multiple variables and ranges for those variables. The presbycusis sets have, for example, type and location of a given SNP and type and result of auditory test. With environmental sampling, examples include the type and count of species at a site, along with environmental factor type and measurement value. Percent leaf litter coverage and average depth, as well as type of forest and type and percent water coverage are examples of the environmental factors and their respective measurements. Since both cases feature hierarchical data sets and multi-dimensionality, they are well-suited for analysis by CRA.

CRA (Cluster Rank Analysis) is one statistical analysis application available for use in analyzing these types of data sets. In performing an analysis, CRA constructs a phylogenetic tree based on a given variable, such as the sequence of the HVR-II, or species count data. The positions of the leaves in a two-dimensional representation of the tree are then swapped based on a secondary variable, such as hearing ability at a given frequency, to maximize the gradient across leaves of the tree; Figure 2 illustrates this. This swapping occurs within the constraints imposed by the topology. In this figure, the gradient is maximized from top to bottom, with the decision to swap any two children based on the value of the two child nodes at a given point. The nodes in question may be two leaves, a leaf and an internal node, or two internal nodes. The values of internal nodes are the mean values of all children, calculated recursively down to the leaves.

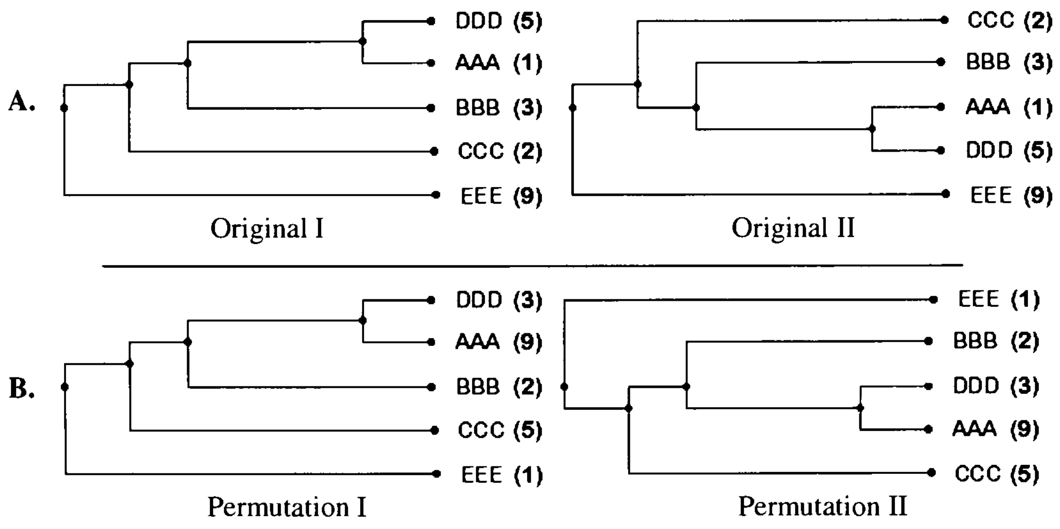


Figure 2. A simplified example of the CRA algorithm. "Original I" shows the tree created, with each leaf node paired with its actual secondary value for this hypothetical secondary data set. "Original II" shows the same tree after swapping has occurred. Note that the tree topology has not changed. "Permutation I" features the same tree with the secondary data values shuffled; "Permutation II" is the resulting tree after swapping has occurred.

The nodes are then given a rank, and the Spearman rank order coefficient is used to quantify the fit to a gradient. This score measures the correlation between the order of leaves in a tree, and variables such as specific hearing abilities (17). Thus, the correlation is determined between the re-ordered secondary variables of the swapped tree, and those variables in their ultimate order. Alternatively, the order of the leaves themselves can be used. For example, in Figure 2A, the series in question are [2,3,1,5,9] and [1,2,3,5,9]; the Spearman rank coefficient between these two sets is 0.7. For Figure 2B, the first series is [1,2,3,9,5] and the second remains the same, yielding a Spearman score of 0.9. This multivariate analysis allows relationships between one primary hierarchical variable and multiple secondary variables to be quantified.

In the case of the presbycusic data, a certain degree of correlation between similar genotypes and certain phenotypic traits can be either inferred or discounted, depending on the Spearman rank coefficient and the p-value generated through the permutations. As for the environmental sampling data, analysis can reveal correlations between the presence of certain species at a sampling site and environmental factors; this would hopefully reveal factors like sensitivity of groups of unrelated species to environmental conditions, whether they are natural or human-influenced.

Original CRA Implementation

The original implementation of CRA was run by executing a single Perl file; this script began by managing the reading of input files and internally preparing the data for analysis. Primary and secondary variables were stored in internal data structures. During the course of execution, several Web Services implemented in Java were called to perform certain functions and transformations on the data (18). For example, one Web Service constructed a hierarchical tree based on aligned DNA sequence data and passed it back to the Perl executable. Another swapped the leaves of the tree and calculated the ranks of the swapped leaves. A third Web Service calculated the Spearman rank coefficient. Some of the Web Services, such as the one that constructed a tree, called external software packages such as PHYLIP (19).

Despite having data sets that may contain a large number of data points, statistical significance is not elucidated simply by the calculation of the Spearman scores; these scores are simply coefficients of correlation between two similar sets that may be ranked differently. CRA must therefore use other means to determine if the actual Spearman scores are statistically significant.

The manner of determining statistical significance used here is to randomly shuffle the secondary data for the primary data points a certain number of times. These shuffled versions of the original tree are then scored, swapped, and compared to the initial Spearman rank coefficients. Thus, the frequency of scores that are at least as strong as the observed, given a comparable random data set, can be determined. For example, if the actual Spearman score for a given secondary variable v is 0.78, and 3 out of 1000 random permutations have Spearman scores better than 0.78, then only 0.3% of shuffled scores for variable v have higher Spearman scores then the original ones. The p-value would therefore be 0.003, less than the typical acceptance threshold of 0.05. The number of permutations performed depends on the precision of the p-value desired. The differences between the first trees in parts A and B of Figure 2 illustrate the shuffling of secondary data values.

After the specified number of permutations had been computed and scored, an HTML table containing the independent variables and the results of their respective permutations was created and written to an output file. Specifically, the original Spearman rank coefficient and number of times that the permuted data yielded a higher Spearman score than the original data were presented.

This original implementation of CRA was designed to run in a fairly distributed manner; much of the computational work was performed by independent Web Services that were called by the Perl executable. This model adhered fairly closely to the service-oriented architecture paradigm. Note that while the term “service-oriented architecture”, or SOA, is often applied incorrectly to any software system utilizing Web Services, the original implementation of CRA came rather close to being a true SOA, according to commonly-accepted definitions of the architecture (20).

A number of processors or machines may have been used in an execution of the original CRA implementation. The client on which the Perl executable resided was machine 0; each machine that a necessary Web Service ran on was machine 1 through *n*, with *n* being the number of independent servers that were hosting Web Services. In the original implementation, *n* could range from one to four, since there were four Web Services called during the course of execution.

Problems with the Original CRA Implementation

In the original decentralized implementation, CRA suffered from several problems. These ranged from performance issues resulting from the number of tree permutations that needed to be created and scored, to human factor issues such as the lack of an intuitive user interface. Additionally, due to its service-oriented architecture, CRA initially suffered from performance limitations based on available network bandwidth.

As described earlier, CRA carries out permutations on hierarchical trees in which it maintains proper linkage within the tree, while determining an order of nodes and leaves that minimizes the Spearman rank order coefficient. The number of permutations that must be computed is typically around 1000, for a p-value precision of 0.001. Creating one permutation and scoring it for a single tree is a trivial process on current computer processors. However, as the number of trees that need to be considered grows, so does the time a given set of processors needs to complete the task. The approximate run-time to fully permute and score a tree of arbitrary size, thus covering every possible permutation of the tree, on a fixed number of processors can be seen in Figure 3. The runtime is *O(n!)*; that is, the computation time grows at a rate that is the factorial of the number of data elements in the set being permuted (21). This notation typically refers to an arbitrary number of computational steps that need to be performed, but can be accepted as a rough estimate of the relative runtime required (22).

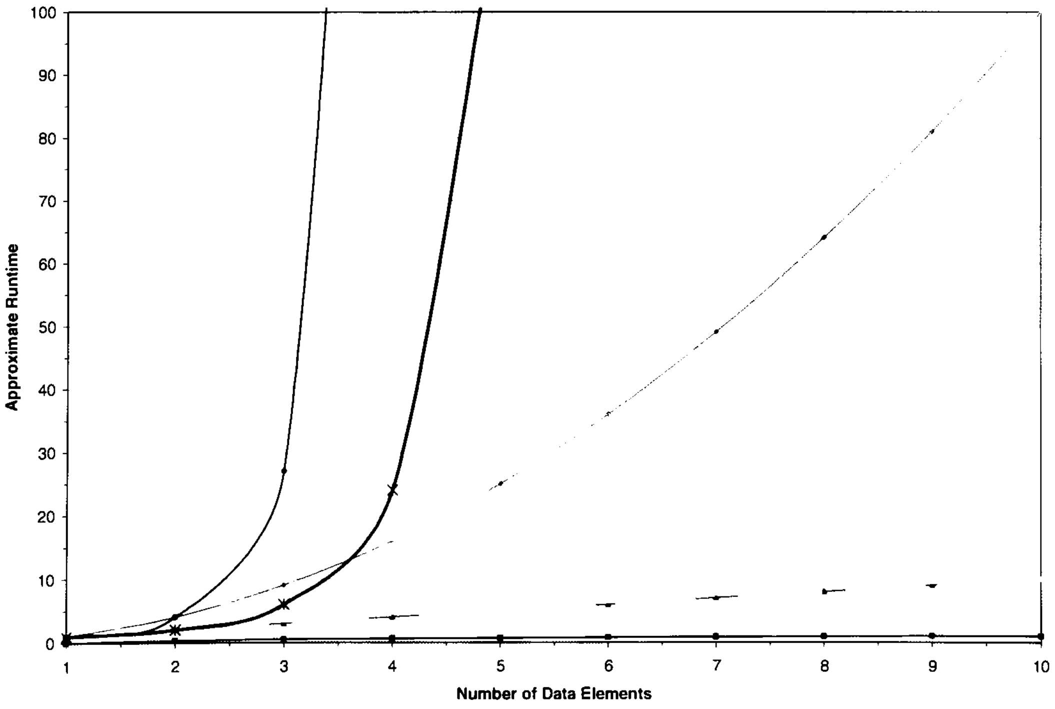


Figure 3. Graph showing the approximate computation time to fully permute a set of elements containing an arbitrary number of data points. Respectively, from fastest-rising to slowest-rising, the curves represent algorithms with *O* values of x^x (●), *x!* (✕), x^2 (◆), x (▲), and *log(x)* (■).

Due to CRA having been implemented as a central Perl "hub" that utilized Web Services, it ran on multiple processors during a typical course of execution. However, despite having had a number of processing units involved during a run of CRA, only one would be performing CRA-related computations at any given time. The client running the main executable had at most one processor dedicated to the application; the same was the case for each of the Web Services, which were not running in parallel on their respective machines. To illustrate this, Figure 4 shows two possible physical architectures for the original version of CRA; one (Figure 4A) assumes that all components of the system, both the executable and the Web Services, are running on a single machine. The other (Figure 4B) assumes that the executable is running on one machine, and each Web Service is running on a unique machine.

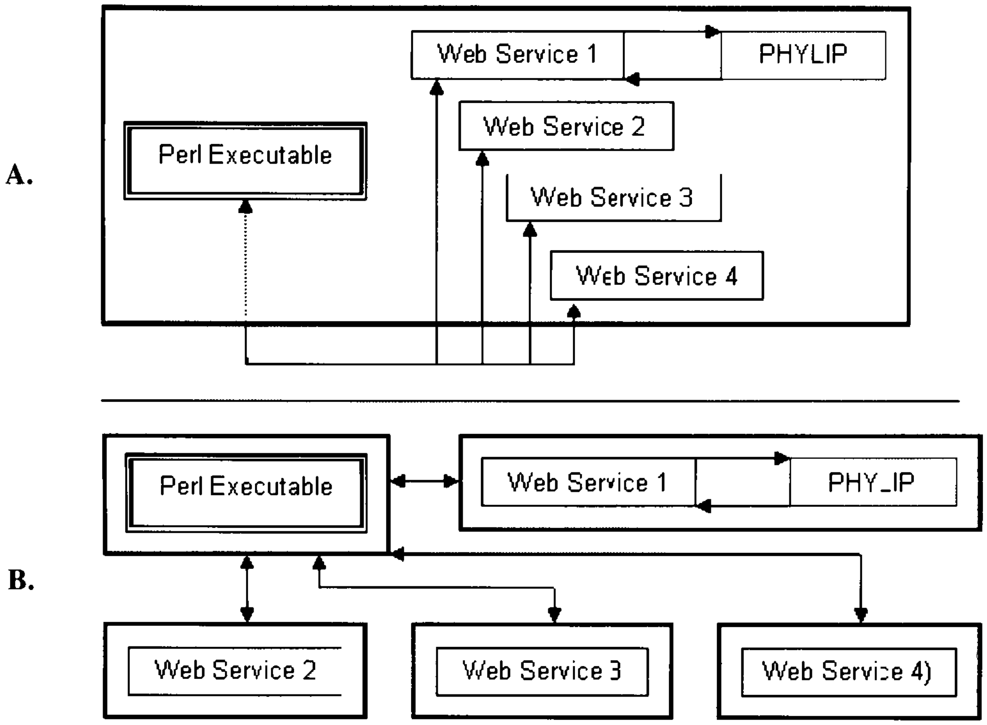


Figure 4. Schematics showing two possible physical architectures for CRA instances. Instance A (top) features both the main executable and the Web Services running on the same server. Instance B (bottom) features the main executable on a separate client from the Web Services; each of the Web Services runs on servers unique to each Service. Instance B also happens to be the logical architecture of the system.

It is important to note that while the physical architecture may have differed between individual CRA instances, the logical architecture was identical to that in Figure 4B; that is, the main executable had no explicit information regarding the physical location of any given Web Service. While Internet protocol (IP) addresses could be used in calling Web Services, the use of uniform resource identifiers (URI's) transfers knowledge of physical addresses to DNS servers; CRA only had to know a logical URL that could then be mapped to any physical location (23, 24). Abstraction such as this is one of the hallmarks of service-oriented architectures (20).

Since each one of the permutations required calls to Web Services in order to be scored and ranked, Ethernet latency was an additional problem that CRA suffered from in its original implementation. Typically, a large number of permutations will need to be made in order to achieve the desired p-value precision; therefore, an equally great number of calls must be made over a network to accomplish this. The main executable and Web Services ran either on the same server, or on servers within an intranet. This helped to reduce the time lost to transmission of data; however, such an approach is not feasible when data must travel over the Internet or over high-traffic intranets. Additionally, each call to a Web Service required a lookup to map the logical address to a physical one. This presented yet another possible bottleneck as more and more independent machines became utilized during an execution. In both sequential and parallel applications, communication overhead is one of the biggest performance barriers to overcome. The effect of communication overhead can be seen in the wide range of times that message transmissions may span. Inter-processor communication via shared memory or high-speed interconnects may take less than a millisecond, while inter-server communication over Ethernet may take as long as several seconds if there is a lot of traffic vying for bandwidth. (25, 26)

An extension to the problem of Ethernet latency and communication overhead is the potential decrease in reliability that is present when multiple independent servers are required to complete an analysis. If a machine hosting a required Web Service is under heavy load from other applications running on it, it may be slow to respond to a request. Even worse, if the aforementioned machine was to fail, the CRA executable would eventually have to abort execution due to a major component of its architecture being unavailable. Since calls to the Web Services were synchronous, the main executable would stop and wait until it received either a response to its request, or an error of some sort (18). Thus, both delay of a response and absence of a response could severely impact the performance of CRA in its original implementation.

The problem of shared resources was introduced above. This refers to servers typically not being dedicated to a given task, whether it is running the main CRA executable or hosting any of the Web Services. Even if communication between two machines was instantaneous, other processes running on both client and server machines could impact the performance of CRA drastically. Application servers are often combined with Web servers and possibly other server setups as well. One instance of CRA ran entirely on the RIT Bioinformatics departmental server, Pastamaker. In this case, Pastamaker was running the Perl executable and hosting and running the Web Services. At the same time, potentially dozens of other users were utilizing CPU cycles. This is a prime example of how conditions outside the scope of a given application, such as CRA, can impact the performance of a program.

One final issue that CRA had was its lack of an intuitive user interface. This was unrelated to overall performance and reliability; however, in order to adopt a program for frequent or mainstream use, users must feel comfortable using an application with the minimum possible amount of training. Since it was executed from a non-CGI Perl script, a user had to have access to a machine that had Perl interpreters installed, as well as a command line interface. The command line interface was the only way to specify run-time parameters and input and output files. The user therefore had to have previous knowledge of the number, type, and order of these arguments, or be able to look at the source code and read the comments within to determine what the parameters should be. As a user base grows, such an implementation could become intimidating and impede the adoption of such a program.

Materials & Methods

General Approach to Converting CRA to a Standalone Application

In making CRA a viable solution for the large-scale multivariate data analysis described previously, many aspects of the program needed to be modified. These modifications consisted of not only the parallelization of the program, but also other usability improvements such as a simplified submission system for running the program and a Web interface for viewing the results of analyses.

A multi-step approach was taken to make the required changes to the program. The first step consisted of converting the Java Web Services to Perl modules. As described earlier, there were initially four Web Services called during a typical execution, with a fifth alternate Service. Thus, five independent Web Services had to be rewritten in Perl. Additionally, certain PHYLIP programs that were called by some of the Web Services, such as tree generation, were converted to Perl, using existing modules wherever possible. The Comprehensive Perl Archive Network (CPAN) was used to locate the appropriate Perl modules, yielding better integration of the program's required components (27). The conversion to Perl modules allows for reusable, modular components; this maintains some of the rationale behind the original service-oriented architecture (20), along with the abstraction that extracting independent portions of code achieves. By having modular code, other applications that might need to take advantage of the methods available in the modules may easily do so. The largest

Perl source repository, CPAN, is a prime example of the widespread acceptance, use, and promotion of modules in Perl software development (27).

Following the conversion of the Java Web Services to Perl modules, the main Perl executable was tested using the modules, as they had to function properly before development could continue. Successful execution of this code, including valid output, allowed development to pass the first checkpoint. The resultant product was a stand-alone, serial version of CRA as a Perl application.

Optimization of CRA

The original version of CRA was, as outlined earlier, converted from a distributed service-based architecture to an integrated application that contains all necessary methods. The runtime of the subsequent serial application was greatly reduced over the distributed system, which was expected considering the problems with public-network Ethernet communication.

In addition to the preexisting Perl modules that are used during execution, there are several CRA-specific modules that were written during the conversion of CRA to standalone form. A distance module (Distance.pm) handles the generation of both DNA and species distance matrices, and any other implemented distance matrix methods. The DNA distance matrix, as mentioned earlier, is computed using BioPerl; the species distance matrix, on the other hand, is computed entirely within this module (28). A tree module (Tree.pm) converts a BioPerl-generated parentheses-delimited tree into a dynamic tree using Perl's built-in data structures. Finally, a swapping module (TreeSwapping.pm) handles the flipping of the leaves and nodes based on their secondary variables, the sorting of the secondary variables, and the traversal of the tree to determine the order of leaves after the flipping.

After this initial stage, several elements of the program were modified further optimize it before parallelization. This ensured that the serial version used for timing was one of the best possible implementations, a necessity in producing reliable speedup data. The first serial version relied rather heavily on object orientation, in both BioPerl and CRA itself. The main benefit of using objects is that the code remains fairly modular and the objects and their methods tend to be previously documented. However, object oriented (OO) functionality is not currently native to Perl, and therefore adds a degree of overhead to applications. The decision was made to sacrifice the cleanliness and modularity of object orientation for the sake of increasing efficiency and decreasing runtime. The non-OO serial version utilizes native Perl data structures such as hashes, arrays, and references, rather than objects.

Parallelization of the Optimized Serial Version of CRA

The second step in the development process was the actual parallelization of the stand-alone application. Development occurred on both a parallel cluster (described below) and sequential machines; the reason for this is that part of the parallelization could be done serially, with occasion testing in parallel. This prevented unnecessary utilization of the Cluster's resources, which are currently used fairly heavily by about a dozen different users. Further testing and tuning during the parallelization process had to be performed in parallel, however. Synchronization of the development between the two types of machines was maintained through the use of modules; rather than having to write the same lines of code on two systems, modules could instead be copied when changes were made.

Parallelization itself was accomplished using MPI (Message Passing Interface), an API for interprocessor communication. This model for parallel computing is used widely, with bindings available for many programming languages (29). As with most parallel programs, execution begins on one processor, with the code branching out onto a specified number of processors. Successful execution in parallel, which again included valid output, allowed this portion of the project to pass the second checkpoint.

The division of the algorithm among an arbitrary number of processors began with the division of permutation calculations among the slave (compute) nodes. The root node created the distance matrix and tree, and swapped and scored that original tree for each of the secondary data points. This approach was then modified to allow the compute nodes to begin working as soon as possible; after the root node finishes creating the tree, it sends the tree and the secondary data to the compute nodes. They can then begin the permutations while the root node scores the original tree and sends those scores to the compute nodes. After the permutations are complete, the scores are compared to the original ones and the results are sent back to the root node and tallied.

To determine the number of permutations that each slave node must compute, the number of permutations is divided by the number of processors being used. Each of these values is then increased by one; this accounts for cases in which the number of permutations and number of processors are not evenly divisible. The user is thus guaranteed to have, at a minimum, the number of permutations computed that they specified.

Computational Resources

The initial conversion of CRA to a standalone application was carried out on serial workstations. After the serial version was finished and tested for proper functionality, further development was performed on the IBM High Performance Computing Cluster (RIT Cluster; the Cluster) (30).

The RIT Cluster, formerly known as plexus.bioinformatics.rit.edu, is a Beowulf cluster of machines housed in the Center for Advancing the Study of CyberInfrastructure and is maintained and administrated by RIT's Research Computing Department (31). The Cluster consists of one IBM xServer 345 head node, which contains dual 2.0GHz Intel Xeon processors and 1Gb of memory. The head node manages the Cluster's network connection to external networks, and is what users log onto when using the Cluster. This node also determines what tasks are assigned to which compute nodes, which will be discussed later in this section.

In addition to the head node, there are forty-seven compute nodes available. Each of these machines is an IBM xServer 330, with dual 1.4GHz Intel

Xeon processors and 512Mb of memory. The compute nodes are connected to each other and the head node through gigabit Ethernet and a high-performance network switch.

The Cluster runs the University of California's ROCKS software package; this includes CentOS Linux as the operating system, the Apache Web server, security software, grid computing packages, system monitoring software, and Sun Microsystem's Sun Grid Engine (SGE) (32, 33). SGE is used to schedule both parallel and serial jobs; a user submits a script with runtime and, if necessary, parallel parameters to SGE, which will then send the job to the appropriate compute nodes when the required number of nodes is available for use. Of note, since the head node is the primary machine and governs the compute nodes, both parallel and serial jobs are run on compute nodes only; this maintains the overall integrity of the Cluster.

Development and Source Code Resources

Development was carried out using the Eclipse Integrated Development Environment, in conjunction with the EPIC Perl extension for Eclipse (34, 35). This extends Eclipse's noted Java development functionality to the Perl language; furthermore, this extension utilizes the PadWalker module. This module allows the contents of allocated memory and variables to be viewed in real-time while debugging; this alleviates the tedious process of having a program print out variables as it runs, which is how debugging in Perl is traditionally accomplished (36).

The BioPerl package is used in several areas of both the optimized serial and parallel versions of CRA. The generation of a distance matrix and creation of a tree rely heavily on modules available in BioPerl. The generation of a distance matrix utilizes the Bio::AlignIO, Bio::Align::DNAStatistics, and Bio::Matrix::PHYLIPDist modules (28, 37). For the creation of a hierarchical tree, Bio::TreeIO, Bio::Tree::DistanceFactory, Bio::Tools::Phylo::Phylip::Prot-Dist, and IO::Capture::Stdout are used (28, 37, 38). Custom distance methods, such as the one used for generating species distances, require an intermediate file between creation of the distance matrix and creation of the tree. The module String::Random is used to give an arbitrary random name to this intermediate file; it is later removed from the file system after the tree is created (39). Finally, Time::HiRes is used to collect the high-resolution times needed for the timing study (40).

Other Perl modules available on CPAN are utilized in the new versions of CRA. The generation of Spearman rank coefficients is handled by a function in an available Perl module; the functions used in the timing study on both versions of CRA are also publicly available (41, 40).

Parallel CRA uses MPI for interprocessor communication. The bindings between the Perl source code and the MPI functions available on the Cluster are managed by MPI.pm, an MPI implementation in Perl (42). The most common MPI functions, such as "Send" and "Recv" (receive), are fully implemented in this module. However, certain functions such as "Bcast" (broadcast) have not yet been implemented, to the detriment of Parallel CRA's performance; this issue will be discussed further.

Several of the aforementioned modules are usually not present in a typical Perl installation; they are quickly and easily installed, though. Most of the remaining functionality in CRA uses standard Perl features that are included with every installation. The other required modules are Distance.pm, Tree.pm, and TreeSwapping.pm; these are the three external modules developed for CRA and are part of the application core.

Evaluating the Performance of Parallel CRA

Following successful parallelization, benchmarking was performed to determine the speedup achieved by the parallel implementation of CRA. (Of note, previous testing to pass checkpoints verified that the results from the parallel implementation were valid and consistent with those from the original sequential implementation.) Benchmarking was performed on the Cluster and compared the rewritten serial CRA implementation to the parallel implementation run both on one processor and then on an arbitrary number of processors. This allowed speedup to be determined, which is defined as (43):

$$S_p = \frac{T_1}{T_p}$$

Here, speedup is S_p, where p is the number of processors currently being used to execute the program, T_1 is the run-time for the original sequential algorithm, and T_p is the time the parallel algorithm takes to run on p processors. Speedup allows the overall improvement in performance to be measured. The numerator here can be substituted with the time it takes the parallel implementation to execute on one processor, thus yielding speedup†; this allows analysis of the parallel version as it runs on 1 to p processors. Speedup†, while not used very frequently in timing studies, can be effective in standardizing timing results between 1 and p processors when there is a fundamental algorithmic difference between serial and parallel versions of a program.

Efficiency was also calculated, which is a measurement of the speedup per processor, and is defined as (43):

$$E_p = \frac{S_p}{p}$$

It can thus be determined how effective each processor is in increasing speedup, whose curve tends to flatten out as the number of processors used increases. This flattening is the effect of Amdahl's Law, which states that the maximum speedup attainable is limited by the portion of a parallel program that must run sequentially or serially (44). Amdahl's Law is demonstrated in Figure 5.

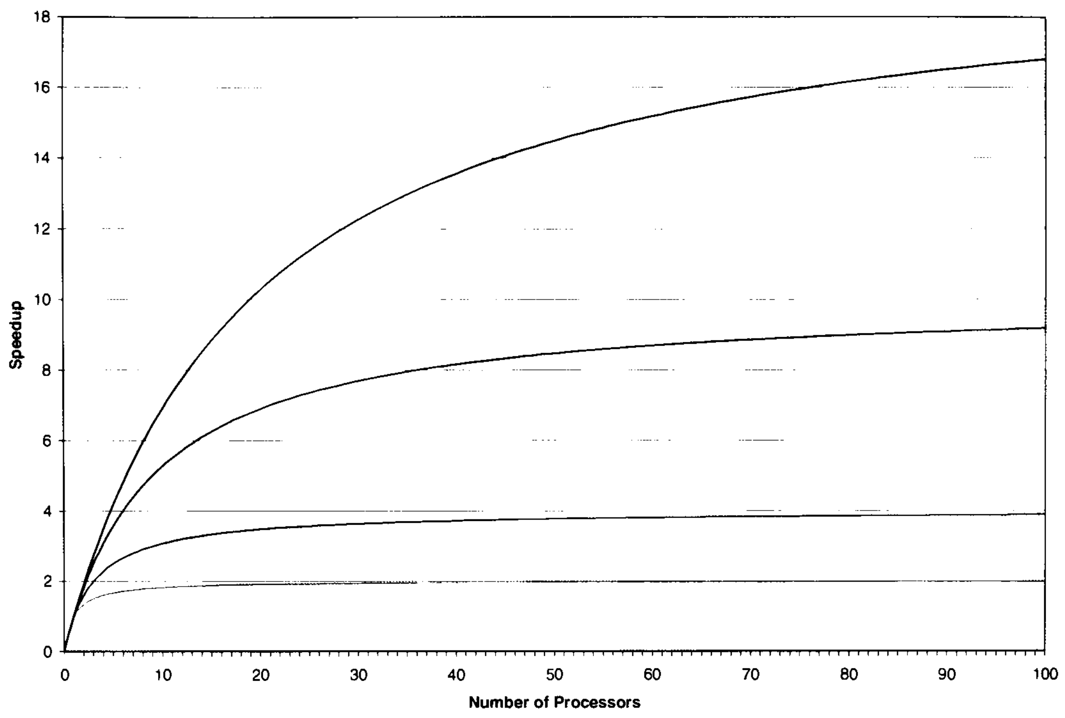


Figure 5. Graph showing the effect of Amdahl's Law on maximum attainable speedup (x axis), which is limited by the fraction of a program that must run sequentially. Curves eventually approach an asymptote that limits the benefit of additional processors on speedup (y axis); this asymptote is the inverse of the percentage of the code that must run sequentially. The curves shown here represent, respectively from top to bottom, programs with 5%, 10%, 25%, and 50% serial run-time.

Finally, the Karp-Flatt metric was applied to the timing data. This metric is often a good indicator of the reason for a limit in speedup. There are two practical reasons for limits of maximum speedup. The first, governed by Amdahl's Law, is that the serial portion of code in a program is the limiting factor, assuming that the parallelization is optimal. The second reason is that there is a perceivable amount of overhead in the parallel implementation, which increases as the number of processors, p, increases.

The Karp-Flatt equation is defined, in practice, as (45):

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

In this equation, p is the number of processors and ψ is the speedup attained given p processors; this version of the Karp-Flatt equation is derived from a theoretical equation which can be interpreted as the serial portion of the parallel algorithm plus all parallel overhead, divided by the runtime on one processor (45).

Development of a Submission System for Parallel CRA Jobs

The final development step was the implementation of a user interface for the setup, execution, and presentation of results from Parallel CRA. This system was originally intended to be a Web interface. However, as will be discussed in *Results*, several obstacles arose that guided the submission of jobs away from a Web interface for the time-being. The retrieval of results from CRA analyses is, however, possible via the Web. While gathering qualitative metrics on the usability and function of such an interface is difficult with such a small user-base, qualitative requirements such as ease of use and robustness guided development here.

A Perl script was created that prompts the user for the names of the files to upload, and various parameters to be submitted to CRA and the queuing system on the cluster. These include: CRA's parameters, such as the type of data that is being submitted and the number of permutations to perform; cluster parameters such as the number of processors to use and the user's username on the cluster;

and job parameters such as the name of the job and the user's email address. Of note, a random string is appended to job names to avoid job name and output file name collisions, as the output file names are based on the original job names (39). The program creates a parameter script that will be used by the cluster's queuing agent to run the job. Fairly robust checks are performed against the information provided by the user, to ensure validity in fields such as email addresses, file names, processor and permutation ranges of values, and usernames.

The verification of usernames is accomplished by checking a configuration file for names which have been previously authenticated by the Cluster. Public-private key DSA authentication is first set up by a modified version of the LUSSH script from the LUFS project (46, 47). This authenticates users once and then allows for subsequent logins *sans* password. The names of users that have done this are appended to a configuration file for use by the submission script.

Following the gathering of parameters, a zip file is created that contains the primary and secondary data files, and the parameter script. The zip file is uploaded to the cluster, unzipped, and submitted to the Cluster's parallel job queue. The zip file is subsequently deleted from the Cluster.

The results of a parallel run are available to the user in a meaningful format upon completion of the analysis. The user is notified by email when their job has finished running; notification is also made when a job starts running and if any errors occur that result in the job being aborted. The resulting HTML file is available to the user on the Cluster, via the Web. Cascading Style Sheets (CSS) are used to format the document, while a publicly available JavaScript (sorttable.js) enables the sorting of HTML tables; this allows the results to be sortable by column in ascending or descending order (48, 49).

The use of email for notifying the user of job status is due to the time span that might exist between submission of a job and start of a job; this latency is a factor of both the number of users utilizing the Cluster's resources at a given time, the number of processors being used by other users, and the number of processors that the user specified CRA to use. Any errors that result in job abortion are typically due to problems on the Cluster, rather than in CRA's source code; however, users should still know if such errors occur, so that they have the option to resubmit those jobs.

Results

Timing Study Setup and Parameters

The optimization and parallelization of CRA resulted in two separate applications. The first is a serial version of CRA that runs on one processor and lacks the overhead associated with the original service-oriented implementation; the second, a parallel version that can run on one or more processors. As described previously, the output values produced by intermediate versions of these programs were compared to known results, to ensure that the changes made in each developmental stage did not cause the program to output erroneous data.

The primary timing study was performed using presbycusis data. The primary data file consisted of mitochondrial DNA sequence data that had been previously aligned and edited; the set contained sequence data for 227 individuals, with each entry being 216 nucleotides in length. The secondary data file contained the results of 40 different hearing tests for each of the 227 subjects involved. The standard number of permutations chosen for timing was 1000; this is the value used in determining a p-value to a precision of 0.001, as described in the *Introduction* with regard to evaluating statistical significance.

Determining Parallel and Serial Runtimes

To determine the runtime of the serial version of CRA, several runs were carried out using the above data at different times on the cluster. This distribution of runs ensured that consistent times were gathered, unaffected by external factors. Once a general estimate could be made of the time that a serial run should take when using the above parameters, a series of five runs was carried out. The mean of these was then taken, resulting in a runtime of 6557.37 seconds (alternatively, 109.3 minutes or 1.82 hours). This value represents T_1, the time required for CRA to run serially, and was used in determining speedup in the parallel implementation. The improvement of the new serial version over the original SOA version of CRA was not quantified, due to the latter not being currently deployed.

After parallelization was completed, a series of parallel runs was carried out. The data sets and number of permutations remained the same; the number of processors, however, varied. Beginning with two processors and increasing in steps of five up to 81 processors, 17 sets of three runs each were performed. As with the determination of the serial runtime, the mean time for each group of three runs was taken to be the runtime for that number of processors. This first series represented ideal runtimes; a second series of runs was carried out, using the above processor groupings in addition to a run using 86 processors, but with two runs per group. The reason that only two runs per processor group were performed is that at the time, the Cluster was scheduled to be taken down for an indeterminate period of time for a physical relocation. The second series featured a number of non-ideal runtimes, illustrating several aspects of running programs in a shared environment; for example, the affect of users running jobs outside an application queuing system.

It was noted that a range from 2 to 86 processors was used; this represents the total number of processors used in a given parallel run. Subsequent tables and figures, however, refer to processors in a range from 1 to 80 or 85; this is due to the approach taken during parallelization. Recall that one processor creates the distance matrix and tree, and scores the initial tree. After that, the permutations are divided up among the remaining non-root processors. For example, if 5 processors are used, processor 1 performs the initial serial work, while processors 2 through 5 will create and score the permutations.

Since the concern of this timing study is the number of processors that the permutations can be divided among, the number of processors performing permutations will be accounted for as p henceforth. Overall, this will not affect S_p, speedup for p processors, but will marginally affect E_p, efficiency for p processors. The runtimes realized in the parallel runs were therefore T_1 through T_{85}.

The average time that each set of runs took to finish is shown below in Table 1, in both seconds and minutes. Timing data for 85 processors in the first run was unattainable due to high cluster usage; the timing data for 85 processors on the second run was acquired in the hours leading up to the shutdown and relocation of the Cluster in early March.

The effect of an increase in processors on the total runtime is better depicted in Figure 6; this figure has two additional curves, which will be explained below.

When these runs were performed, data was also captured relating to the portion of the program that was not parallelized. In this case, loading the primary and secondary data, creating a distance matrix, and constructing a tree are the portions of code that run serially; the permutations are then distributed among the slave processing nodes.

Table 1. The number of processors used and total runtimes for the first and second series of CRA runs in parallel. $T_p[1]$ and $T_{pMIN}[1]$ are for the first series of runs, in either second or minute notation, while $T_p[2]$ and $T_{pMIN}[2]$ are second and minute values for the second run. Note that these processor counts are for the number of processors performing permutations. The total number of processors involved in each run was equal to $p+1$.

p	$T_p[1]$	$T_{pMIN}[1]$	$T_p[2]$	$T_{pMIN}[2]$
1	6348.54	105.81	6381.34	106.36
5	1501.96	25.03	1560.97	26.02
10	899.25	14.99	916.50	15.27
15	695.44	11.59	885.60	14.76
20	591.14	9.85	719.10	11.98
25	532.88	8.88	609.28	10.15
30	489.63	8.16	531.58	8.86
35	462.18	7.70	504.30	8.41
40	441.97	7.37	522.89	8.71
45	424.69	7.08	449.99	7.50
50	412.34	6.87	481.80	8.03
55	402.11	6.70	448.13	7.47
60	390.36	6.51	437.83	7.30
65	384.27	6.40	383.91	6.40
70	376.56	6.28	374.07	6.23
75	371.20	6.19	372.15	6.20
80	364.39	6.07	363.88	6.06
85	**	**	439.85	7.33

The component of the serial portion that took a non-negligible amount of time to complete was the construction of the distance matrix from the primary data. Therefore, the time that this took to complete was taken to be the serial portion of the code, a mean value of 254 seconds, or about 4.23 minutes, for the data and parameters used.

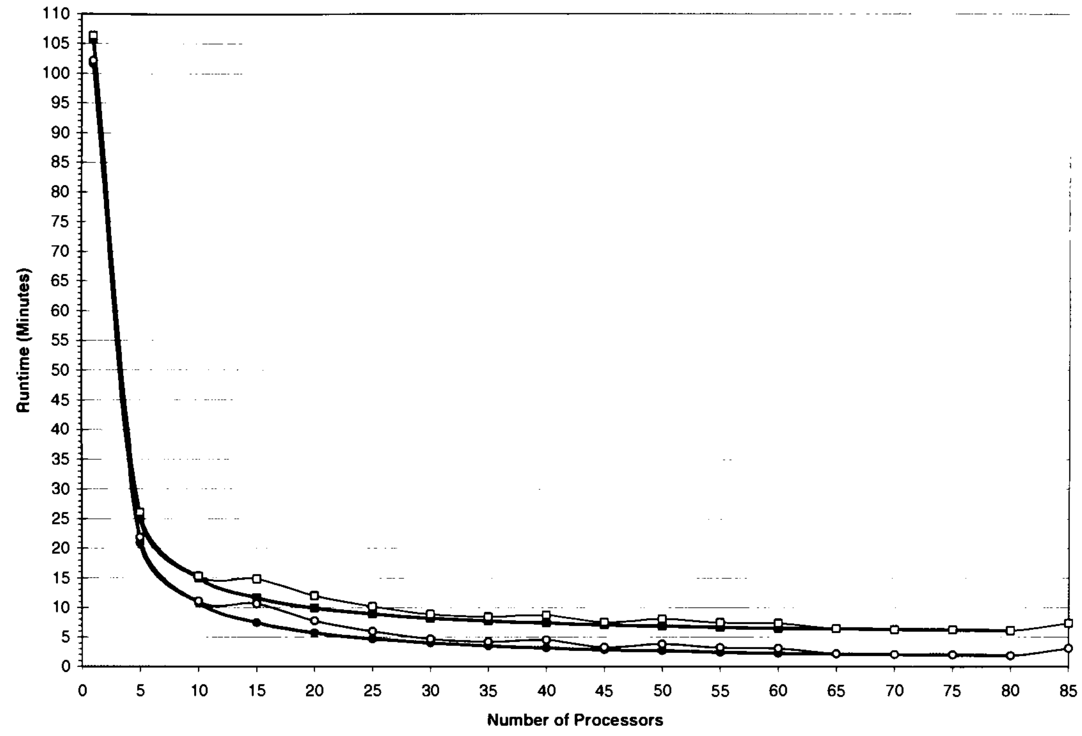


Figure 6. Graph showing the reduction in runtime as the number of permutation-performing processors is increased. The top two curves represent the total runtimes for, respectively from top to bottom, the second (□) and first (■) timing series. The lower two curves, also respectively from top to bottom, represent the time that the second (O) and first (●) runs required solely to calculate and score 1000 permutations.

The time required to construct the distance matrix was then subtracted from the total time that each CRA run took. This allowed an analysis of the effectiveness of the parallelized portion of code to be performed; the runtime analysis of which can be seen in the lower two curves in Figure 6. These curves are based on the runtime values for permutations found in Table 2; the lack of data for 85 processors in the first timing run was previously explained. If two data sets use distance matrix methods whose runtimes differ materially, the subtraction of the serial runtime allows for a more equal comparison between metrics such as speedup; this is due to multiple data types using the same algorithm for generating and scoring permutations. As explained in *Materials & Methods*, several different types of distance-matrix-generating algorithms may be used by implementing them in the appropriate module.

Table 2. The number of processors used and permutation runtimes for the first and second series of CRA runs in parallel. $T_{pPERMS}[1]$ and $T_{pPERMS\text{-}MIN}[1]$ are for the first series of runs, in both second and minute notation, while $T_{pPERMS}[2]$ and $T_{pPERMS\text{-}MIN}[2]$ are for the second series of runs.

p	$T_{pPERMS}[1]$	$T_{pPERMS\text{-}MIN}[1]$	$T_{pPERMS}[2]$	$T_{pPERMS\text{-}MIN}[2]$
1	6098.75	101.65	6129.30	102.16
5	1251.00	20.85	1309.23	21.82
10	646.69	10.78	663.89	11.06
15	444.67	7.41	634.32	10.57
20	340.77	5.68	463.28	7.72
25	280.22	4.67	358.37	5.97
30	237.46	3.96	279.05	4.65
35	207.18	3.45	251.01	4.18
40	189.43	3.16	271.37	4.52
45	170.56	2.84	194.91	3.25
50	159.00	2.65	227.07	3.78
55	146.92	2.45	193.16	3.22
60	134.67	2.24	183.02	3.05
65	128.59	2.14	128.30	2.14
70	122.53	2.04	122.57	2.04
75	116.43	1.94	116.36	1.94
80	110.29	1.84	110.42	1.85
85	**	**	183.21	3.05

Speedup Achieved in Parallel CRA

Speedup values, presented in Table 3, ranged from 1.07 to 18.68 in the first series of runs, and from 1.07 to 18.71 in the second series of runs. Recall that speedup is calculated for p processors as T_1/T_p; T_1 is the mean time for the serial version when calculating ordinary speedup, and the mean time for the parallel version running on one processor for the seldom-used speedup†. Unless otherwise noted, "speedup" here is referring to the former of the two- ordinary speedup.

Table 3. The number of processors used and overall speedup (S_p) values for the first, ideal series ($S_p[1]$) and second series ($S_p[2]$) of CRA runs in parallel, along with speedup values based only on the permutation runtimes ($S_{pPERMS}[1]$ and $S_{pPERMS}[2]$ for the first and second runs, respectively). As there was no timing data for 85 processors in the first series of runs, there are no corresponding speedup values available.

p	$S_p[1]$	$S_p[2]$	$S_{pPERMS}[1]$	$S_{pPERMS}[2]$
1	1.07	1.07	1.08	1.07
5	4.53	4.36	5.24	5.01
10	7.57	7.43	10.14	9.88
15	9.79	7.69	14.75	10.34
20	11.52	9.47	19.24	14.15
25	12.78	11.17	23.40	18.30
30	13.90	12.81	27.61	23.50
35	14.73	13.50	31.65	26.12
40	15.40	13.02	34.62	24.16
45	16.03	15.13	38.45	33.64
50	16.51	14.13	41.24	28.88
55	16.93	15.19	44.63	33.95
60	17.44	15.55	48.69	35.83
65	17.72	17.73	51.00	51.11
70	18.08	18.20	53.52	53.50
75	18.34	18.29	56.32	56.36
80	18.68	18.71	59.46	59.39
85	**	15.48	**	35.79

The speedup values presented above are depicted graphically in Figure 7. The diagonal represents "ideal speedup", which is the theoretical maximum speedup achievable given the fastest-possible serial algorithm and a perfect parallel version of the same algorithm. The lower curves in this figure, which represent overall speedup from the timing runs, along with the extended trend line, indicate that speedup is tending toward a maximum of approximately 19.

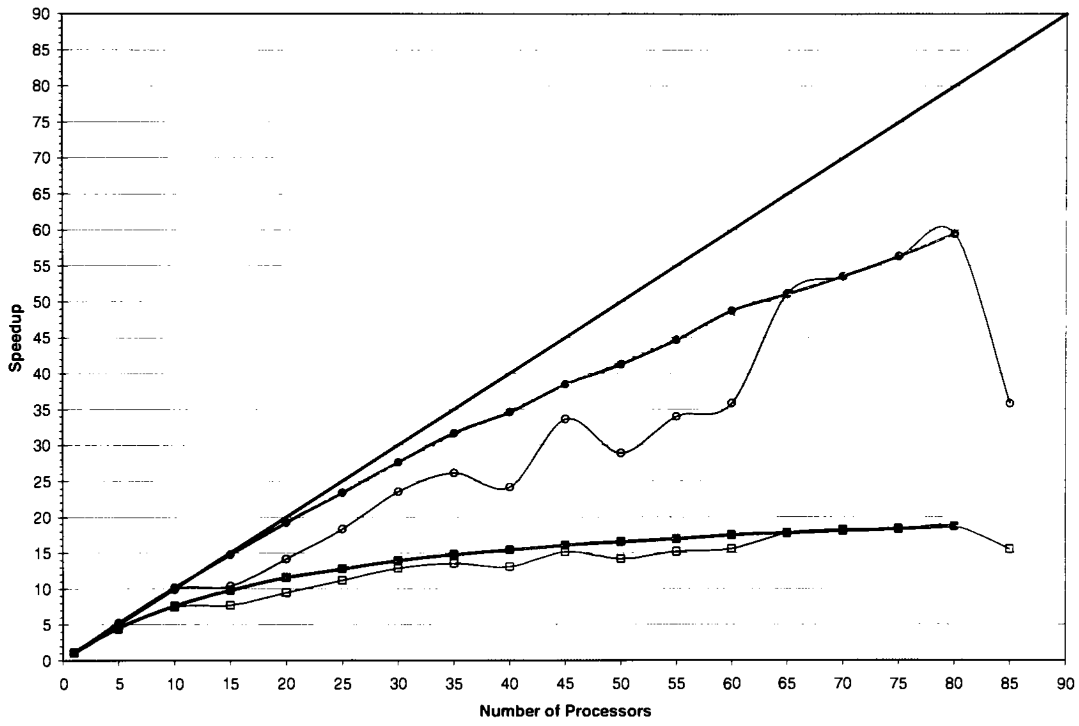


Figure 7. Graph showing the increase in speedup as the number of permutation-performing processors is increased. The diagonal represents the ideal, or maximum possible, speedup. The top bold curve (●) represents permutation-based speedup for the first series of timing runs. The lower bold curve (■) represents overall speedup for the first series. The thin curves below each of these represent the respective speedup values from the second set of timing runs (O for permutation-based, □ for overall). Extended trend lines for the ideal timing series are shown as dotted lines.

The upper two curves in the figure represent the permutation-based speedup results for the two timing series. Since these curves are based solely on the parallel portion of the program, they are likely to have no maximum value provided that sufficiently large data sets are used. In practice, however, the increase in speedup would eventually be limited by the number of available processors.

Timing fluctuations in the second series of runs, as described earlier, led to the fluctuations in the two speedup curves for this series. It should not be assumed, nor is it likely, that speedup drops off so sharply at 85 processors. Furthermore, the permutation-based speedup curve continues to grow up to 80 processors. Even if a speedup curve were to drop, it should plateau before a decline occurs.

Efficiency Achieved in Parallel CRA

The analysis of speedup can be extended to look at efficiency, a measure of each processor's contribution to the speedup value. This is defined as S_p/p, literally the speedup per processor given p processors. Table 4 presents the efficiency values for the two timing runs. In addition to the overall efficiency, the permutation-based analysis is continued; the $E_{p\mathrm{PERMS}}$ values detail how much each additional processor contributes to the permutation-based speedup found in Table 3.

Table 4. The number of processors used and overall efficiency (E_p) values for the first series ($E_p[1]$) and second series ($E_p[2]$) of CRA runs in parallel, along with efficiency values based only on the permutation runtimes ($E_{p\mathrm{PERMS}}[1]$ and $E_{p\mathrm{PERMS}}[2]$ for the first and second runs, respectively).

p	$E_p[1]$	$E_p[2]$	$E_{p\mathrm{PERMS}}[1]$	$E_{p\mathrm{PERMS}}[2]$
1	1.07	1.07	1.08	1.07
5	0.91	0.87	1.05	1.00
10	0.76	0.74	1.01	0.99
15	0.65	0.51	0.98	0.69
20	0.58	0.47	0.96	0.71
25	0.51	0.45	0.94	0.73
30	0.46	0.43	0.92	0.78
35	0.42	0.39	0.90	0.75
40	0.39	0.33	0.87	0.60
45	0.36	0.34	0.85	0.75
50	0.33	0.28	0.82	0.58
55	0.31	0.28	0.81	0.62
60	0.29	0.26	0.81	0.60
65	0.27	0.27	0.78	0.79
70	0.26	0.26	0.76	0.76
75	0.24	0.24	0.75	0.75
80	0.23	0.23	0.74	0.74
85	**	0.18	**	0.42

These efficiency values are illustrated in Figure 8. The two bold curves are based on the first series of timing runs, with the upper one comprising the permutation-based values and the lower one comprising the overall efficiency values. The thin curves consist of data points from the second timing series.

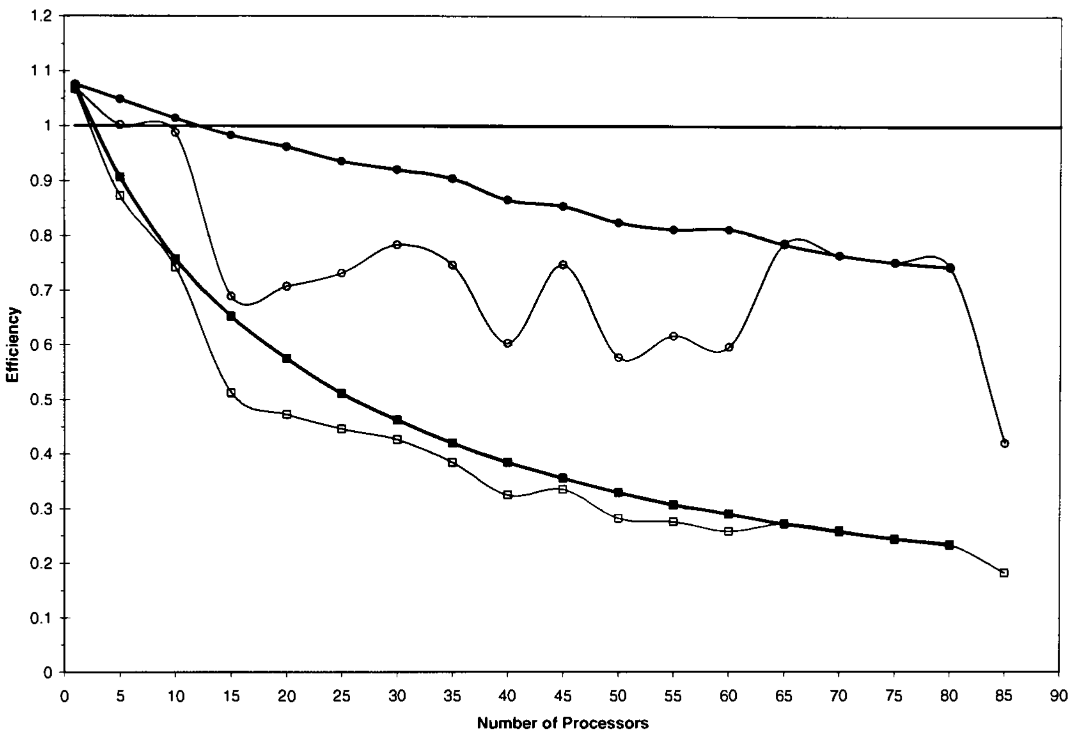


Figure 8. Graph showing the decrease in efficiency as the number of permutation-performing processors is increased. The horizontal line at y=1 represents ideal efficiency. The top bold curve (●) represents permutation-based efficiency for the first series of timing runs. The lower bold curve (■) represents overall efficiency for the first series. The thin curves below each of these represent the respective efficiency values from the second set of timing runs (O for permutation-based, □ for overall).

Discrepancies in Runtime, Speedup, and Efficiency Values

In several previous figures, the large variance present in the second series of runs is a result of the fluctuations in runtimes for that series; the same is the case here. Recall that this sub-ideal data has been included to illustrate both the shortcomings of computational resource-sharing and the effect that poor timing results have on other performance metrics that are derived from runtimes.

Several of the initial data points in Figure 8 fall above the theoretical maximum efficiency of 1, represented by the line y=1; additionally, several of the values found in Table 3 are greater than their respective number processors, an indication of super-linear speedup. This is a result of the method of parallelization, and will be explained in greater detail in the *Discussion.* The region of super-linear speedup is shown in Figure 9. Note that the curve for the permutation-based speedup of the first timing run is above the diagonal beginning at the start of the series, and continuing until shortly after p=12 processors.

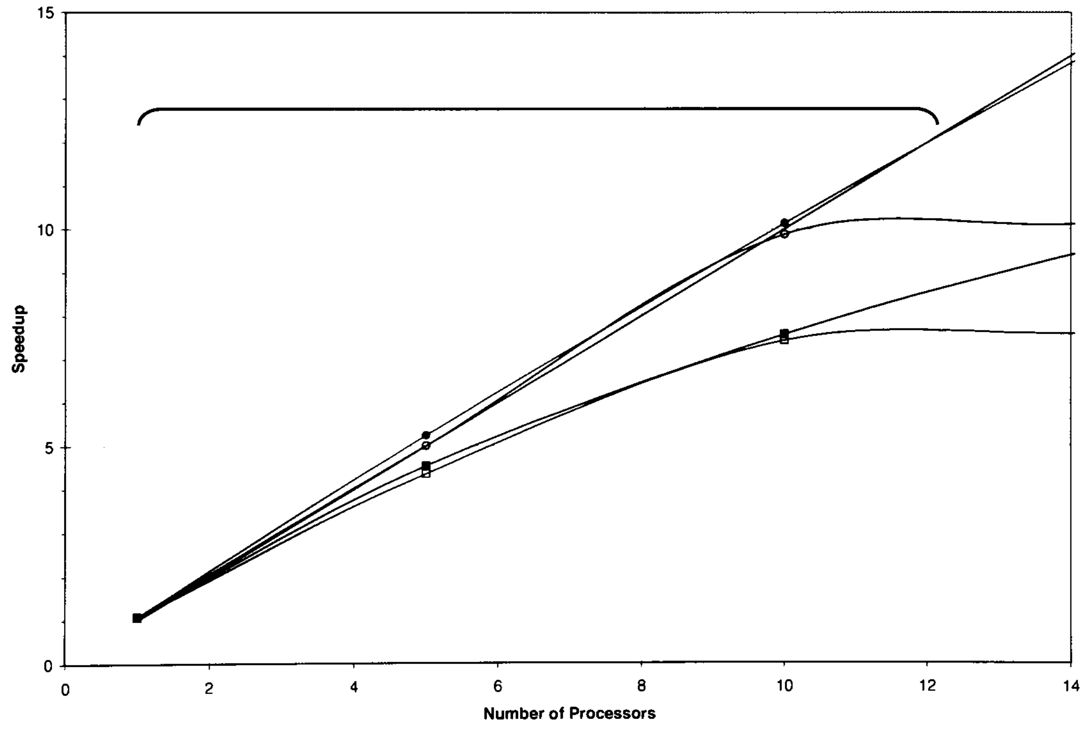


Figure 9. Graph showing the region of super-linear speedup for the two timing runs. The top two curves represent permutation-based speedup for the first (●) and second (O) series of timing runs. The lower two curves represent overall efficiency for the first (■) and second (□) series of runs. The horizontal bracket near the top indicates the region of super-linear speedup for the first timing series, from p=1 to approximately p=12.

Correlations Among Performance Metrics

The effect that runtime can have on speedup can be seen in Figure 10. For example, note that the relatively small increase in runtime at p=85 led to a moderate drop in overall speedup at that value of p, and a tremendous drop in permutation-based speedup; this is a result of the extra runtime occurring during the parallel portion of execution.

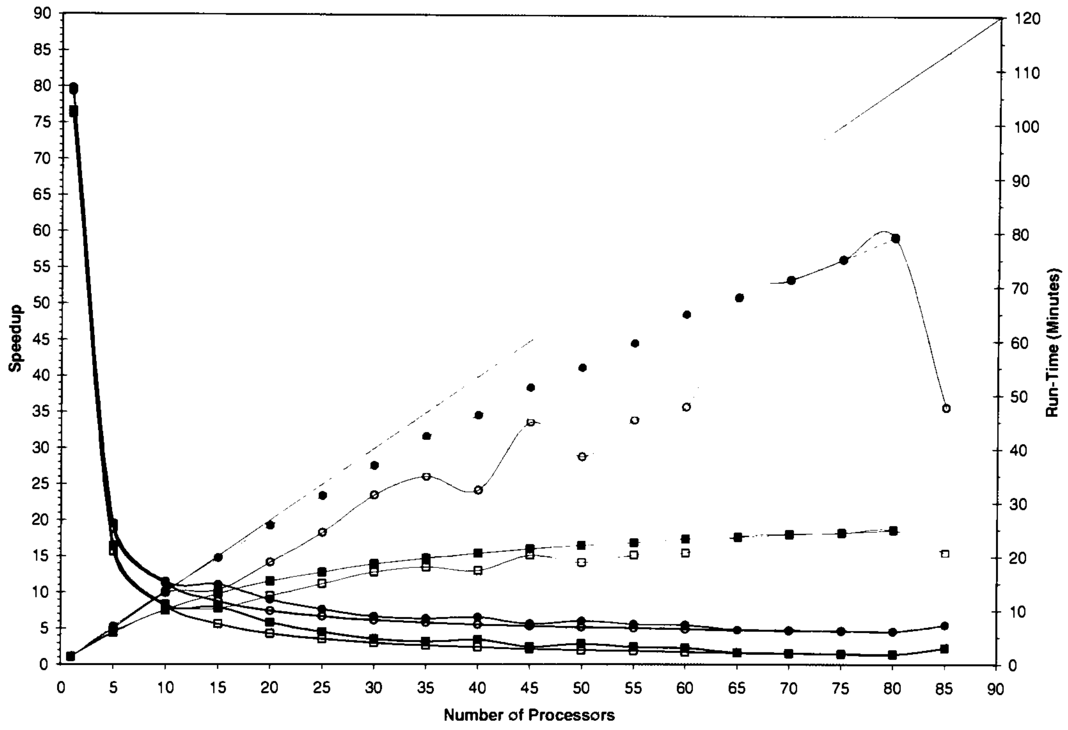


Figure 10. Graph overlaying Figure 6 and Figure 7, allowing the correlation between runtime and speedup to be clearly seen. The nomenclature for the curves remains the same, with the overall values for the first and second series of runs marked by (■) and (□) , respectively, and permutation-based points marked by (●) and (O) for the first and second series, respectively. The series that begin above the diagonal are the runtime curves, while those beginning at or below the diagonal are the speedup curves.

The effect the speedup has on efficiency is illustrated in Figure 11. Here it is demonstrated how decreases in speedup are met with decreases in efficiency, which become decreasingly greater as the number of processor increases.

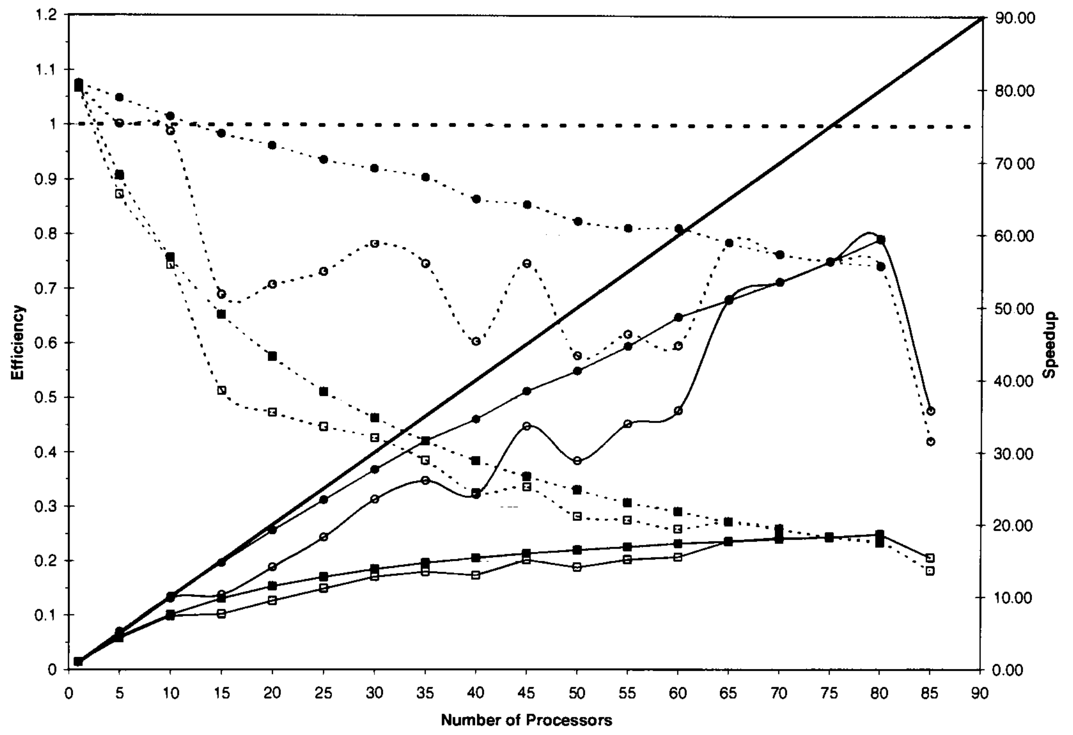


Figure 11. Graph overlaying Figure 7 and Figure 8, allowing the correlation between speedup and efficiency to be seen. The nomenclature for the curves remains the same, with the overall values for the first and second series of runs marked by (■) and (□), respectively, and permutation-based points marked by (●) and (O) for the first and second series, respectively. The series that begin above the diagonal are the efficiency curves, while those beginning at or below the diagonal are the speedup curves.

Karp-Flatt Metric for Evaluating Parallelization

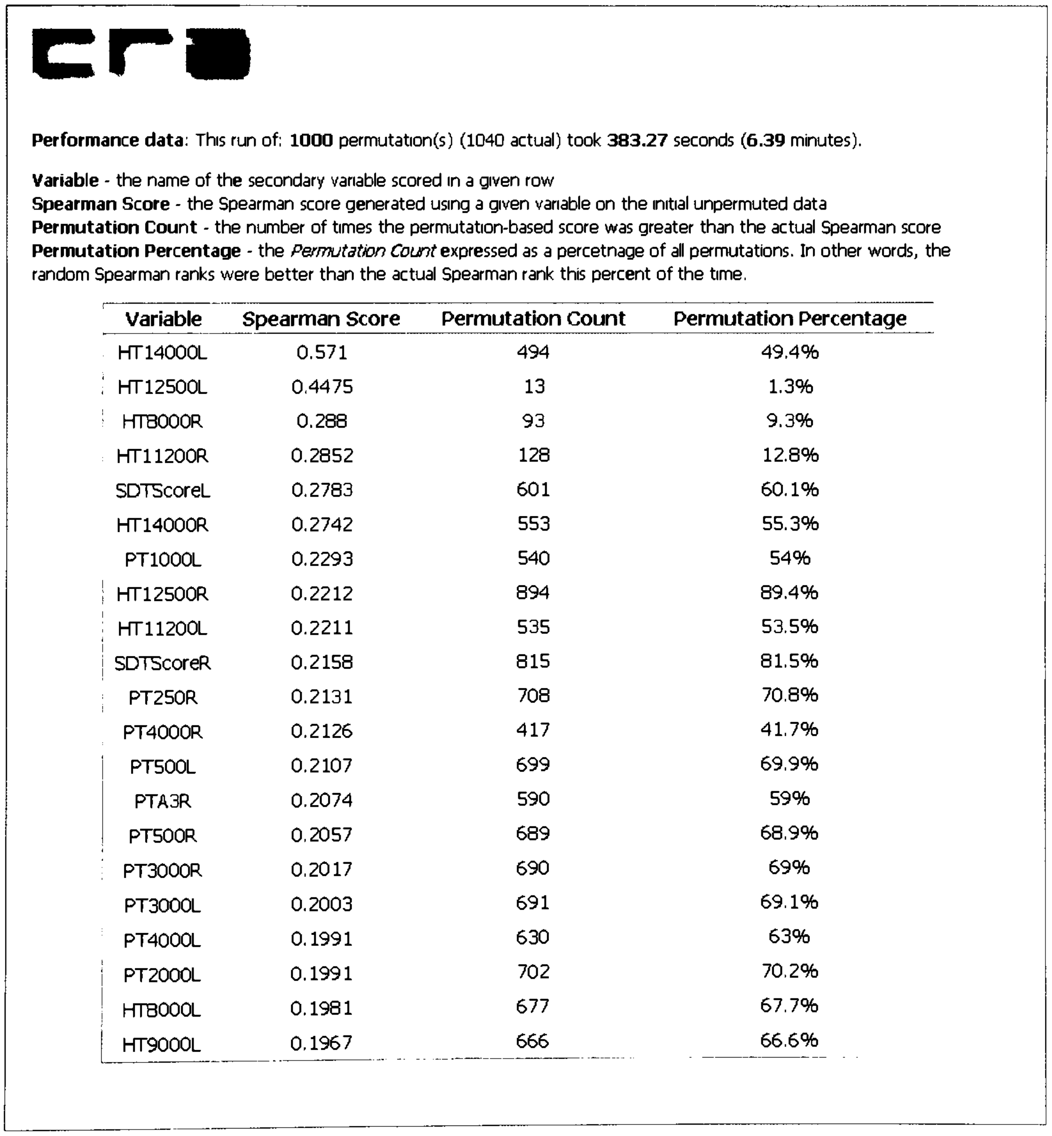
The final performance metric that was considered with regard to the parallelization of CRA was the Karp-Flatt metric. As noted earlier, this metric is useful in determining whether the limiting factor in the speedup attained is a large serial portion of code, or overhead related to an increase in the number of processors. Table 5 shows the Karp-Flatt values, commonly referred to as *e*, for both of the timing runs. The values of *e* in the first timing run increase slightly over the first several runs, but then stabilize to approximately 0.4. For the second run, the values of *e* did not necessarily increase steadily as *p* increased; however, there was much greater variance in these *e* values, with a range of 0.37 to 0.68.

Table 5. The number of processors used and overall Karp-Flatt (*e*) values for the first ($e_p[1]$) and second ($e_p[2]$) of timing runs.

p	$e_p[1]$	$e_p[2]$
5	0.026	0.037
10	0.036	0.038
15	0.038	0.068
20	0.039	0.059
25	0.040	0.052
30	0.040	0.046
35	0.040	0.047
40	0.041	0.053
45	0.041	0.045
50	0.041	0.052
55	0.042	0.049
60	0.041	0.048
65	0.042	0.042
70	0.042	0.041
75	0.042	0.042
80	0.042	0.041
85	**	0.053

Web-Based Interface for Retrieval of CRA Results

The layout and design of a typical CRA results page can be seen in Figure 12. As mention in *Materials & Methods*, the table of results is sortable in ascending or descending order by any of the available columns. The user is also given some performance data regarding the runtime, along with brief explanations of what the table headers represent.



Performance data: This run of: **1000** permutation(s) (1040 actual) took **383.27** seconds (**6.39** minutes).

Variable - the name of the secondary variable scored in a given row
Spearman Score - the Spearman score generated using a given variable on the initial unpermuted data
Permutation Count - the number of times the permutation-based score was greater than the actual Spearman score
Permutation Percentage - the *Permutation Count* expressed as a percetnage of all permutations. In other words, the random Spearman ranks were better than the actual Spearman rank this percent of the time.

Variable	Spearman Score	Permutation Count	Permutation Percentage
HT14000L	0.571	494	49.4%
HT12500L	0.4475	13	1.3%
HT8000R	0.288	93	9.3%
HT11200R	0.2852	128	12.8%
SDTScoreL	0.2783	601	60.1%
HT14000R	0.2742	553	55.3%
PT1000L	0.2293	540	54%
HT12500R	0.2212	894	89.4%
HT11200L	0.2211	535	53.5%
SDTScoreR	0.2158	815	81.5%
PT250R	0.2131	708	70.8%
PT4000R	0.2126	417	41.7%
PT500L	0.2107	699	69.9%
PTA3R	0.2074	590	59%
PT500R	0.2057	689	68.9%
PT3000R	0.2017	690	69%
PT3000L	0.2003	691	69.1%
PT4000L	0.1991	630	63%
PT2000L	0.1991	702	70.2%
HT8000L	0.1981	677	67.7%
HT9000L	0.1967	666	66.6%

Figure 12. Screenshot of a typical results page for a CRA run. The table is sortable by any of the four column headers, simply by clicking on them.

Discussion

Outcome of Running CRA in a Shared Environment

The optimization and parallelization process was quite successful. The first timing run demonstrated consistently shorter runtimes with each increase in the number of permutation-performing processors. The second timing run also had, in general, a decrease in runtime with each subsequent addition of processors. However, it was noted that this second run was sub-optimal and included to demonstrate several drawbacks to running applications in shared environments.

As Figure 6 shows in the difference between the upper and lower curves for both timing runs, the time to generate the distance matrix remained relatively constant over the series. Thus, the fluctuations in runtime at 15, 40, 50, 55, and 85 processors during the second timing series are not due to any load imbalance on the root node. The additional runtime then had to come from either a bottleneck in interprocessor communication time or from processor slowdowns on the compute nodes.

The timing runs were supervised using the Ganglia Monitoring System for high-performance computing systems (50). This allowed the tracking of metrics such as network bandwidth, processor loads, and active processes. None of the timing runs resulted in heavy bandwidth usage, nor was the system's communication network being strained by other parallel jobs. It was observed that certain users were running parallel jobs outside of the parallel queue. The Sun Grid Engine (SGE), which handles the queuing of parallel processes, schedules jobs such that the load is balanced across available processors. Unfortunately, it is only aware of jobs that it started, and not processes that users began themselves via terminal sessions. Some users were thus permitted to usurp entire compute nodes, with SGE assuming that these nodes were not being utilized at all; analyzing the processes and loads on individual compute nodes showed this to be the definitive cause of errant performance. Multiple processes were then vying for 100% of available CPU cycles, causing a massive slowdown in the time it took to create and score permutations. The runtime fluctuations in the second series of timing runs were due to this. The effect on the generation and scoring of permutations can be seen in Figure 7; the overall speedup curves were affected only slightly by increases in runtime; however, the permutation-based speedup curves are heavily impacted by runtime increases. These slowdowns were found in the parallel portion of CRA, so those curves are lacking the buffer of serial runtime that the overall speedup curves have. The policies for using the Cluster should, in the future, address this issue to ensure that non-trivial jobs are not permitted to run outside of SGE.

Optimum Balance of Processors in Parallel CRA

The runtime curves in Figure 6 suggest that while any increase in processor count will result in a decrease in runtime, it may not be necessary to try to use every available processor. After a certain point, around 15-20 processors for the timing runs here, the decreases in runtime became marginal. If the cluster was a single-user environment, then utilizing every available processor would be optimal. However, since other users are typically competing for processors, it is most acceptable to use as few processors as needed for a job. Considering the fact that jobs take more or less 5-10 minutes to run with 20-85 processors, it may be better to allow fellow users the other 60+ processors and accept the slight performance reduction.

Additionally, total "submission-to-results" time may be significantly greater when more processors are being used; this is due to SGE waiting until all of the desired processors are available before it executes a job. A user may spend several hours waiting for 70 processors to become available, only to save 5 minutes of runtime, which is imprudent; 20 processors may be immediately available, leading to results much less time. Of course, the optimum number of processors needed to balance total runtime and "submission-to-results" time will very depending on the size of the data sets that CRA is analyzing.

Effect of Amdahl's Law on Parallelization of CRA

Amdahl's Law, as previously described and illustrated in Figure 5, essentially limits the speedup possible in a parallel algorithm by the portion of code that runs serially. It was determined, and presented in Figure 6, that the serial portion of execution in the timing runs performed here took approximately 4.23 minutes to complete; recall that this portion was comprised of reading both data files, initializing necessary variables, creating a distance matrix, and producing a hierarchical tree from this matrix. These tasks collectively, with the exception of distance matrix generation, required a negligible amount of time, so the average distance-matrix-generation step was considered to be the serial runtime and minimum possible runtime. The $T_{p\text{PERMS-MIN}}$ values in Table 2 show that as p surpasses 30 to 40 processors, this serial runtime represents over 50% of the total runtime. In fact, at 80 processors, this serial time is approximately 70% of the total runtime. It is estimated that given another 30-50 processors, the time to calculate and score the permutations for the timing set here would reach approximately 1.2-1.5 minutes, leaving the serial runtime comprising 73%-78% of the total runtime (data not shown). This percentage would increase only marginally, if at all, beyond that number of processors.

It might be assumed that a substantial number of processors would result in the serial runtime eventually comprising 99%-100% of the total runtime, as per Amdahl's Law. This is not the case here, with this percentage reaching a hypothetical limit of maybe 80% (data not shown). There are several factors at play here that prevent the serial runtime from assuming nearly the entire run's duration. It should first be noted that the average time to calculate and score each permutation in the timing set was slightly greater than 6 seconds (data not shown). Given 1000 processors, 1000 permutations would not take a total of 6 seconds to calculate and score. The time per permutation may still be a little over 6 seconds, but the overall time for the entire parallel portion of code to execute might take significantly longer than this. The factor behind this, which is also the first of the additional factors to limit speedup here, is the overhead associated with parallel code execution.

Communication Overhead and Sub-Optimal MPI Bindings

It was noted earlier that speedup in parallel problems typically levels out as Amdahl's Law takes effect, and may actually start to decrease as additional processors are incorporated due to additional overhead. In the parallel API used here, MPI, there is overhead associated with initializing and finalizing processors before and after the parallel code to be run is distributed. This usually takes a constant amount of time for any number of processors. Most non-trivial parallel algorithms also feature interprocessor communication. This is where overhead and communication latency can increase as more and more processors are utilized. The Perl MPI module used here has several drawbacks which, in short, resulted in all communication being point-to-point; that is, the root node had to initialize and clean up communication streams between every slave node it sent data to or received data from.

The MPI bindings for Perl, found in the module MPI.pm, implement most of the commonly used MPI functions and constants, such as the ability to send and receive data between two processors, and the ability to determine how many processors are available to communicate with. According to the documentation for MPI.pm, the function "Bcast," which broadcasts a message from one processor to many others simultaneously, has been implemented (42). The tests performed to verify proper compilation of module, however, failed during every installation attempt. These tests were performed after compilation, and the module was actually usable for just about all of the functions that are listed in the documentation as being supported.

It was found that the "Bcast" test kept failing; this was verified by comparing two test scripts; one featured sequential direct communication between the root node and several other processors, and the other featured a simultaneous general broadcast from the root to all other processors. The first run had a single integer value as the message being sent, and both communication methods worked. However, once alphanumeric data was used, the broadcast version of the script failed consistently. It is unknown why the "Bcast" binding does not function correctly when using alphanumeric data, and is not a documented software bug. It would be of interest to persons with intimate knowledge of the MPI framework to determine what causes one data type to succeed and another to fail; additionally, since this MPI module is fairly new and still under development, it would be wise for the aforementioned parties to submit a hypothesis or solution to the developers of MPI.pm.

There is some contention as to whether point-to-point or multipoint communication should be used for interprocessor communication; there are even custom methods available that are more efficient than the native ones (51, 52). It is generally agreed upon, however, that actual one-to-many or many-to-many transactions should utilize a collective communication method. This allows the parallel framework (in this case, MPI) to handle the initialization and breakdown of multiple interprocessor streams. This is most likely at least as efficient as a series of point-to-point calls, due to MPI doing the work that a series of higher-level program calls would otherwise do. A further comparison between the current "Send/Recv" implementation of CRA and one using a functional "Bcast" would be particularly interesting; even if the actual execution time did not decrease, at least the source code would be cleaner.

During parallel execution, the root node sends two messages to the compute nodes, and the compute nodes send two messages to the root node; this seemingly results in twice as much parallel overhead as necessary, especially considering that each send has a matching receive as well. The reason for this is that the amount of data being analyzed is certainly not constant. Metadata variables such as variable names or a change in the number of primary or secondary variables could result in a drastic change in the size of the data packages that need to be sent and received. In MPI, each send and receive action is accompanied by a value that tags the message with the size of the data structure contained within. This is easy to determine for the sender, as it needs only to determine the length of a string. The receiver, however, has no idea how large the message is; however, MPI states that the receive method call is provided a variable which contains the minimum size of the data that has yet to be received. In an environment of constant message sizes or slight fluctuations in message size, a constant value which contains a certain amount of padding can be provided, to ensure that adequate memory is allotted before the message is received. This approach cannot be taken here though, due to the uncertainty in message size.

The solution used here is to first send an integer to the receiving nodes; integers can scale greatly without a large change in the actual number of characters in the number. This integer represents the length of the alphanumeric data message that will be sent. The aforementioned approach, using a fixed value that includes an adequate buffer, is first used to notify the slave nodes of the size of the data message. The receive call can then be provided with the exact size of the data message that will be received, ensuring that only as much memory as needed is allocated on each node. The same two-step approach is taken when the results of the permutation creation and scoring are finished, and the results are sent back to the root node.

Reasons for Super-Linear Speedup and Efficiency

The parallelization of CRA features a slight overlap in execution between the root node and the compute nodes. Since this occurs for only a short period of time, the root node was not included in the processor counts found throughout the *Results* discussed here. This exclusion of a node did not affect the actual speedup values, since the number of processors being used does not factor into the speedup equation. However, the speedup values are still marginally inflated for any given processor count, as one extra processor was actually used for a short period of time during the execution.

Consider Table 3, in which speedup at p = 1 is 1.07; speedup for exactly one processor should ideally be 1, however, indicating that the parallel implementation of an algorithm is neither better nor worse than the serial version when running on one processor. The use of the root processor at the same time as a single slave processor in this case, even for the short period in which the original tree is scored, contributes to speeding up the computation by 0.07; while this is a rather insignificant amount, any degree of super-linear speedup in a parallel algorithm warrants an explanation. Since speedup tends to become decreasingly greater as more processors are added to a problem, it can be assumed that the original factor of 0.07 becomes even less as p increases; thus, the performance increase from the use of an additional processor eventually becomes negligible.

The horizontal bracket in Figure 9 highlights the region of super-linear speedup. The two timing series featured super-linear overall speedup at $p = 1$; however, the permutation-based speedups for these timing runs were super-linear at $p = 1$, $p = 5$, and $p = 10$ for the first timing series and at $p = 1$ and $p = 5$ for the second timing series. The permutation-based speedup values are slightly inflated, as they are based on an inflated permutation-based runtime from the serial version of CRA. This serial value was calculated by subtracting the time required to generate the distance matrix from the total serial runtime; thus the time required to swap and score the original tree was included in the permutation-generating time. In the parallel version, however, the scoring of the original tree happens after parallel execution begins; the permutation-based runtime in these runs truly is the time during which the program creates and scores permutations.

The inflated T_1 permutation-based runtime is only off by the several seconds that it takes to score the original tree using each dependent variable's independent values (data not shown), and the goal of analyzing the permutation-based performance metrics was not necessarily to get absolute values for these; rather, the point was to look at these values relative to one another as the number of processors involved in execution increased. Therefore, while values such as permutation-based speedup may not be absolutely accurate, they are relatively accurate among one another when p is greater than 1; the purpose of looking at these values relative to one another is discussed below.

The use of an additional processor for a portion of execution affects efficiency as well; this is why the overall efficiency values found in Table 4 are greater than the theoretical maximum of one when $p = 1$; however, as with speedup, this efficiency boost will become marginal as the number of processors increases.

Limiting Factors of Maximum Performance Gains

It was determined earlier that the maximum speedup achieved in this study was 18.71; as mentioned, the maximum speedup given additional processors is thought to be somewhere around 19. The two reasons for a limit in speedup, to iterate, are that the serial portion of an algorithm's execution has assumed the majority of the program's runtime, or that the overhead associated with additional processors is increasing and hindering any additional increase in speedup. The problem then arose of determining which of these two reasons caused the speedup in Parallel CRA to be limited; of course, it could also be a combination of both of these reasons. To achieve this, the permutation-based performance metrics were calculated, as was the Karp-Flatt metric.

Calculating the permutation-based runtimes was accomplished, as described earlier, by subtracting the non-parallel runtime from the total runtime for the parallel runs, leaving only the time the permutations took to create and score. Speedup and efficiency were determined using these parallel runtimes and the non-distance-matrix-generating runtime from the serial runs; this resulted in the skew that was described above. These speedup values are based solely on the parallel portion of code in CRA; therefore, any limit based on the serial portion of code has been eliminated. In Figure 7, the relevant curve for the first timing series follows the ideal linear curve quite closely until 30-35 processors are involved. Comparing this to the overall speedup curve (for series 1) in the same figure, it can be seen that by 10-15 processors the latter has begun to slow in ascent. This indicates a speedup limit caused by serial execution for lower processor counts. Since the permutation-based curves (barring runtime irregularities for the second timing series) begin to slow in ascent as higher processor counts are reached, it can be assumed that parallel overhead such as initialization of communication streams then begins to limit speedup further. Amdahl's Law still applies at these higher processor counts; even if the parallel overhead was non-existent, a limit in speedup would still eventually be reached.

Efficiency, illustrated in Figure 8, drops off quickly at first before taking on a progressively slower descent. This metric is essentially the percent utilization of a given processor during a program's execution (43). This figure thus shows the increase in the percentage of time the serial portion of code takes (shown in the overall efficiency curves), as well as the increase in the time that parallel overhead takes, (shown in the permutation-based efficiency curves), causing processors to be utilized increasingly less in execution of Parallel CRA as overhead takes increasingly more time.

The second of the methods used to determine the cause for a limit in speedup is the examination of the Karp-Flatt metric, also known as e; this was described in *Materials & Methods.* A fairly constant value of e indicates that speedup is limited by the serial portion of code, while a growing e indicates an increase in parallel overhead as the cause of limited speedup. The values of e presented in Table 5 suggest that both reasons for limited speedup are present. In the first timing series, there is a slight increase in e over the first few runs, but it then becomes fairly constant. Parallel overhead increases such that the Karp-Flatt metric detects it, but the overhead itself remains fairly constant as the processor count increases; the largest factor limiting speedup is therefore the serial portion of Parallel CRA's execution. The second timing run has a slightly greater range of values, due to the fluctuations in that timing series; however, the average e follows the pattern found in the first timing series fairly closely.

Since the speedup in Parallel CRA is limited largely by the serial portion of the algorithm, it is critical to note that the maximum speedup is therefore linked to the distance-matrix-generating method. This method, which is the primary component of the serial portion of the program, can vary depending on the type of data that is being analyzed. A distance matrix whose construction takes less time than the one used in this timing study will cause the overall runtime to decrease and the speedup to increase, in addition to allowing a higher maximum attainable speedup. If the distance matrix method being utilized takes more time than that used here, the maximum attainable speedup will be less than 19, due to a larger portion of non-parallelizable runtime.

Web-Based Submission and Results for CRA Jobs

The original intent for the submission system was to have a Web-based interface in which users could upload data and runtime parameters to the cluster. The steps following this are as implemented: the cluster runs the job, notifying the user at the start and finish of the run, and then makes the results available via a Web browser. Several obstacles arose, however, that steered the submission system in the direction of an intuitive command line interface.

The administrators of the RIT Cluster, the Department of Research Computing, had the sole requirement that access be no less secure through the Web interface than through terminal sessions, and there was a desire to avoid the creation of "tiers" of users, in which users of the Cluster create "Web users" of their own. Having users of CRA be registered users of the cluster was not a problem in itself, as registering is as simple as filling out an online form. However, users would then either have to have CRA installed in their own directories, or be authenticated to use a common CRA installation.

Having users install CRA in their own directories was ruled out, and the latter solution was considered. Two possibilities presented themselves, the first of which was to authenticate users within CRA. This was ruled out due to the implementation and testing of a custom authentication process being outside the scope of this project. The second possibility was to use existing Apache modules to handle security.

There is a module that allows for the authentication of users against the system user directory, by the Apache Web server itself; however, this creates a security hole in which attempts to crack the system's root password can be made (53, 54, 55). Attempting to disallow the use of "root" as a username would bring back the custom authentication problem mentioned above. Apache will allow for a more secure authentication of users against LDAP; unfortunately, such a directory is not currently available on the Cluster (56, 57). The implementation of LDAP, Globus, or some other universal access system is currently being considered by the Cluster's administrative team; once an acceptable solution is found and implemented, the development of a Web-based submission system will make for an acceptable undergraduate research project (58). The framework is already in place with the command line interface developed here; one must simply provide an acceptable front-end and work with existing Apache modules to grant Web access to system users.

Conclusions

The goal for this study was threefold: the first was to optimize and parallelize the original implementation of the Cluster Rank Algorithm; the second, to perform a timing analysis of the rewritten applications; the third, to implement a more user-friendly data submission and analyses retrieval system for Parallel CRA jobs.

CRA was originally implemented using a service-oriented architecture, which was not necessarily the optimal solution for multiple and frequent exchanges of complex data types. Custom models for data analysis, as well as models capable of managing large and complex data structures, are becoming more practical and more popular as hardware and software platforms become more amenable to approaches such as parallel computing. The model used in CRA is no exception; here a two-dimensional data structure based on one data set is analyzed using a second data set, the equivalent of working with a three-dimensional data structure.

The timing study accompanied the parallelization effort as a requisite step in determining just how much more quickly and how much more efficiently Parallel CRA runs over its serial counterpart. The estimated maximum speedup of 19 is a reasonable figure, albeit difficult to benchmark against other programs due to the uniqueness of most parallelization approaches. As was seen in the permutation-based speedup and efficiency values, as well as the Karp-Flatt metric, the serial portion of the algorithm was the primary limiting factor in speedup; parallel overhead played a much smaller role. It is therefore important to remember that the maximum achievable speedup will vary depending on how long a given distance matrix method takes to execute.

An algorithmic extension to this work, which would increase speedup and efficiency, and decrease runtimes further, would be to parallelize the generation of a distance matrix. This would be sensitive to the overall magnitude of the dependent data, however; it might be quicker to use a serial distance matrix approach for smaller data sets, and utilize a parallel approach above a determined threshold.

The original goal of developing a Web interface to CRA was determined to be out of the scope of this work due to security concerns; this stresses the need for an Apache Web server module to handle authentication against an operating system's user base that eliminates the existing security holes. Until a user directory system is implemented on the Cluster, the command line submission system will have to suffice. This is certainly not appropriate in allowing all users to be able to submit their own jobs. It is, however, far more convenient for the person or persons handling job submission to supply the runtime and output parameters and have a script handle the uploading of data to the Cluster, the creation of a parameter file for the queuing system, and the actual submission of the job to the grid engine. Further development will most likely focus on the implementation of a Web-based submission tool, following the setup of a user directory system on the Cluster; the existing command-line system will provide the framework for this. The goal of being able to retrieve the results of CRA runs from the Web has been realized (59).

Not long ago, parallel computing was a niche market available to large corporations, governments, and anyone else able to afford a supercomputer from Cray or Thinking Machines. The emergence of commodity high-performance hardware and open source software is bringing the efficiency and speed advantages of parallel computing to a much larger user base. This is allowing developers not only to write programs directly for parallel architectures, but also to easily rewrite and optimize programs for parallel systems as they become more widely available. These resources may even be arrays of CPU's housed in a single case, or publicly available pay-per-use systems (60, 61). Parallel CRA demonstrates how high-performance computing resources can be utilized in a relatively short period of time to more quickly and more efficiently analyze large or complex data sets.

References

1. Seidman, M., Ahmad, N., and Bai, U. Molecular mechanisms of age-related hearing loss. (2002) *Ageing Research Reviews*, **1:3**, 331-343.

2. Gates G., Caspary D., Clark W., Pillsbury H., Brown, S., and Dobie, R. Presbycusis. (1989) *Otolaryngology - Head and Neck Surgery*, **100:4**, 266-271.

3. Cilento, S., Norton, J., and Gates, G. The effects of aging and hearing loss on distortion product otoacoustic emissions. (2003) *Otolaryngology - Head and Neck Surgery*, **129:4**, 382-389.

4. Gates, G., and Mills, J. Presbycusis. (2005) *The Lancet*, **366:9491**, 1111-1120.

5. Gates, G., Couropmitree, N., Myers, R. Genetic associations in age-related hearing thresholds. (1999) *Otolaryngology--Head & Neck Surgery*, **125:6**, 654-659.

6. Giles, R., Blanc, H., Cann, H., and Wallace D. Maternal inheritance of human mitochondrial DNA. (1980) *Proc. Natl. Acad. Sci.*, **77:11**, 6715–6719.

7. Newman, N. Hereditary Optic Neuropathies: From the Mitochondria to the Optic Nerve. (2005) *American Journal of Ophthalmology*, **140:3**, 517.e1-517.e9.

8. Lodish, H., Berk, A., Zipursky, S.L., Matsudaira, P., Baltimore, D., and Darnell, J.E. *Molecular Cell Biology,* 4th ed. (1999). New York: W H Freeman & Co.

9. Castora F., Arnheim N., and Simpson, M.. Mitochondrial DNA polymorphism: evidence that variants detected by restriction enzymes differ in nucleotide sequence rather than in methylation. (1980) *Proc. Natl. Acad. Sci.*, **77:11**, 6415–6419. [PubMed]

10. Brandon, M.. Lott, M., Nguyen, K., Spolim, S., Navathe, S., Baldi, P., and Wallace, D. MITOMAP: a human mitochondrial genome database--2004 update. (2005) *Nucleic Acids Research*, **33** (Database Issue):D611-613.

11. Newman, D., Raish, K., Edsall, L., Witkowski, C., Eddins, D., Frisina, D., Shipman, P., and Osier, M. Cluster Rank Analysis provides evidence for mitochondrial inheritance of presbycusis. (2005) 55th Annual Meeting of the American Society for Human Genetics. Salt Lake City, UT.

12. Price, S., Marks, D., Howe, R., Hanowski, J., and Niemi, G. The importance of spatial scale for the conservation and assessment of anuran populations in coastal wetlands of the Western Great Lakes, USA. (2004) *Landscape Ecology*, **20**, 441-554.

13. Savage, J. The geographic distribution of frogs: patterns and predictions. (1973) *Evolutionary Biology of the Anurans.* University of Missouri Press, Columbia. Vial, J.L., ed. 351-446.

14. Shipman, P., Fox, S., Thill, R., Bently Wigley, T., and Melchiors, M.A. Anurans are associated with intermediate landscape-level habitat features in the Ouachita Mountains, Arkansas, USA. (2006) Submitted to *Forest Ecology and Management.*

15. Pearman, P. Correlates of Amphibian Diversity in an Altered Landscape of Amazonian Ecuador. (1997) *Conservation Biology*, **11**, 1211-1225.

16. Guldin, J. Landscape-scale research in the Ouachita Mountains of West-Central Arkansas: General study design. In Ouachita and Ozark Mountains Symposium: Ecosystem Management Research, 143-145. October 26-28, 1999, Hot Springs, Arkansas. J.M. Guldin, (ed.), General Technical Report SRS-74. Asheville, NC: Southern Research Station, USDA Forest Service.

17. Spearman, C. The proof and measurement of association between two things. (1904) *American Journal of Psychology*, **15:72**, 101.

18. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. Web Services Architecture. (2004) W3C. (http://www.w3.org/TR/ws-arch/).

19. Felsenstein, J. 1993. PHYLIP (Phylogeny Inference Package) version 3.5c. Distributed by the author. Department of Genetics, University of Washington, Seattle.

20. Ferguson, D., and Stockton, M. Service-oriented architecture: Programming model and product architecture. (2005) IBM Systems Journal, **44:4**. (http://www.research.ibm.com/journal/sj/444/ferguson.html).

21. Knuth, D.E. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. Section 1.2.11: Asymptotic Representations. Addison-Wesley (1997), 107–123.

22. Knuth, D.E., Big Omicron and Big Omega and Big Theta. (1976) SIGACT News, **8(2)**, 18-24.

23. Mockapetris, P.V., RFC 1034: Domain names - concepts and facilities (Nov. 1987) Status: Standard. Internet Engineering Task Force (IETF).

24. Mockapetris, P.V., RFC 1034: Domain names - implementation and specification (Nov. 1987) Status: Standard. Internet Engineering Task Force (IETF).

25. Cypher, R. and Konstantinidou, S. Bounds on the Efficiency of Message-Passing Protocols for Parallel Computers. (1996) *SIAM J. Computing*, **25:5**, 1082-1104.

26. Abandah, G.A. and Davidson, E.S. Modeling the communication performance of the IBM SP2. 10th International Parallel Processing Symposium (IPPS'96), April 1996.

27. CPAN: Comprehensive Perl Archive Network. (http://www.cpan.org/)

28. Stajich, J.E., Block, D., Boulez, K., Brenner, S.E., Chervitz, S.A., Dagdigian, C., Fuellen, G., Gilbert, J.G.R., Korf, I., Lapp, H., Lehvaslaiho, H., Matsalla, C., Mungall, C.J., Osborne, B.I., Pocock, M.R., Schattner, P., Senger, M., Stein, L.D., Stupka, E.D., Wilkinson, M., and Birney, E.. The Bioperl Toolkit: Perl modules for the life sciences. (2002) *Genome Research.* **Oct;12(10)**, 1161-8.

29. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. (1994) University of Tennessee, Knoxville, TN.

30. RIT Research Computing. RIT Cluster & Cluster User Guide. (2007) Rochester Institute of Technology, Rochester, NY. (http://cluster.rit.edu)

31. Beowulf.org. What makes a cluster a Beowulf?. Beowulf Project Overview. Beowulf.org. (2007) (http://www.beowulf.org/overview/index.html)

32. Papadopoulos, P.M., Katz, M.J., and Bruno, G. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. (October 2001), Cluster 2001: IEEE International Conference on Cluster Computing. (http://www.rocksclusters.org)

33. Gentzsch, W. Sun Grid Engine: Towards Creating a Compute Power Grid. (2001) 1st International Symposium on Cluster Computing and the Grid, 35. (http://www.sun.com/software/gridware/)

34. Eclipse Integrated Development Environment. The Eclipse Foundation. (2007). (http://www.eclipse.org/)

35. EPIC – Eclipse Perl Integration. (http://e-p-i-c.sourceforge.net/index.html)

36. Houston, R. PadWalker - play with other peoples' lexical variables. Comprehensive Perl Archive Network. (2007) (http://search.cpan.org/dist/PadWalker/PadWalker.pm)

37. Hoon, S. and Stajich, J. Bio::Matrix::PhylipDist - A Phylip Distance Matrix object. Comprehensive Perl Archive Network. (2007) (http://search.cpan.org/~birney/bioperl-1.4/Bio/Matrix/PhylipDist.pm)

38. Reynolds, M. and Morgan, J. IO::Capture::Stdout - Capture any output sent to STDOUT. Comprehensive Perl Archive Network. (2005) (http://search.cpan.org/~reynolds/IO-Capture-0.05/lib/IO/Capture/Stdout.pm)

39. Pritchard, S. String::Random Perl module to generate random strings based on a pattern. Comprehensive Perl Archive Network. (2006) (http://search.cpan.org/~steve/String-Random-0.22/lib/String/Random.pm)

40. Wegscheid, D., Schertler, R., Hietaniemi, J., and Aas, G. Time::HiRes - High resolution alarm, sleep, gettimeofday, interval timers. Comprehensive Perl Archive Network. (2007) (http://search.cpan.org/~jhi/Time-HiRes-1.9707/HiRes.pm)

41. Boggs, G. Statistics::RankCorrelation - Compute the rank correlation between two vectors. Comprehensive Perl Archive Network. (2007) (http://search.cpan.org/~gene/Statistics-RankCorrelation-0.10/lib/Statistics/RankCorrelation.pm)

42. Wilmes, J., and Stevens, C. Parallel::MPI An MPI Binding for Perl. Rensselaer Polytechnic Institute. 9 February, 2007. (http://search.cpan.org/~josh/Parallel-MPI-0.03/MPI.pm)

43. Eager, D.L., Zahorjan, J., and Lozowska, E.D. Speedup Versus Efficiency in Parallel Systems. (1989) *IEEE Transactions on Computers.* **38:3**, 408-423.

44. Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. (1967) American Federation of Information Processing Societies Spring Joint Computer Conference.

45. Karp, A.H. and Flatt, H.P. Measuring Parallel Processor Performance. (1990) *Communication of the ACM,* **33:5**, 539-543.

46. U.S. Department of Commerce and National Institute of Standards and Technology. Digital Signature Standard, FIPS PUB 186-2. (2000). Federal Information Processing Standards Publication.

47. Malita, F. Lufs Userspace Filesystem Framework. (2003) (http://lufs.sourceforge.net/lufs/)

48. Bos, B., Çelik, T., Hickson, I., and Lie, H.W. Cascading Style Sheets, level 2 revision 1. CSS 2.1 Specification. (2006). World Wide Web Consortium (W3C). (http://www.w3.org/TR/2006/WD-CSS21-20061106/)

49. Langridge, S. sorttable: Make all your tables sortable. (2007) (http://kryogenix.org/code/browser/sorttable/)

50. Massie, M.L., Chun, B.N., and Culler, D.E. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. University of California, Berkeley. Technical Report. (February 2003).

51. Gorlatch, S., Send-receive considered harmful: Myths and realities of message passing. (2004) *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **26:1**, 47-56.

52. Mateescu, G. A Method for MPI Broadcast in Computational Grids. (2005) In *Proceedings of the 19th IEEE international Parallel and Distributed Processing Symposium (Ipdps'05)*, Workshop 13, Volume 14 (April 04 – 08, 2005).

53. Apache Software Foundation. Apache HTTP Server Project. (2007) (http://httpd.apache.org/)

54. Neulinger, N., Allison, T., and Wolter, J. Mod_Auth_External and Mod_Authnz_External – Apache External Authentication Modules. (2007) (http://unixpapa.com/mod_auth_external.html)

55. Wolter, J. pwauth – A Unix Web Authenticator. (2006) (http://www.unixpapa.com/pwauth/)

56. Wahl, M., Howes, T., and Kille, S. Lightweight Directory Access Protocol (v3). (1997) Internet Engineering Task Force, Request for Comments (IETF RFC). RFC 2251.

57. Apache Software Foundation. mod_auth_ldap - Allows an LDAP directory to be used to store the database for HTTP Basic authentication. Apache HTTP Server Version 2.0 (2007) (http://httpd.apache.org/docs/2.0/mod/mod_auth_ldap.html)

58. The Globus Alliance. Globus Toolkit. (2007) (http://www.globus.org/)

59. Esposito, A.G. CRA Results Page. (2007) (http://cluster.rit.edu/CRA/)

60. Sun Microsystems. Sun Utility Computing. (2007).
(http://www.sun.com/service/sungrid/index.jsp)

61. Microway, et al. (2007) (http://www.microway.com)